Operating System Principles

操作系统原理

Process&Thread Concepts

李旭东

leexudong@nankai.edu.cn

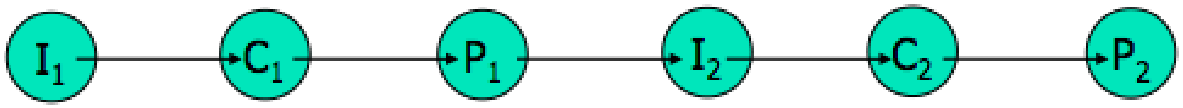
Nankai University

Objectives

- Execution of Program
- Process Concept
- Thread Concept

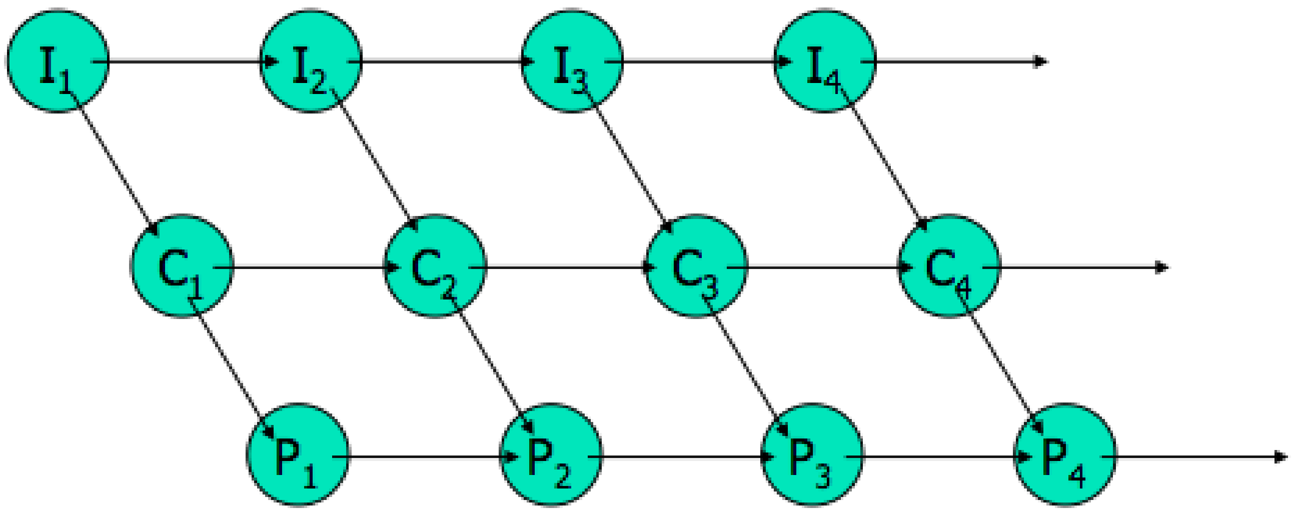
Execution of Program

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



Execution of Program

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



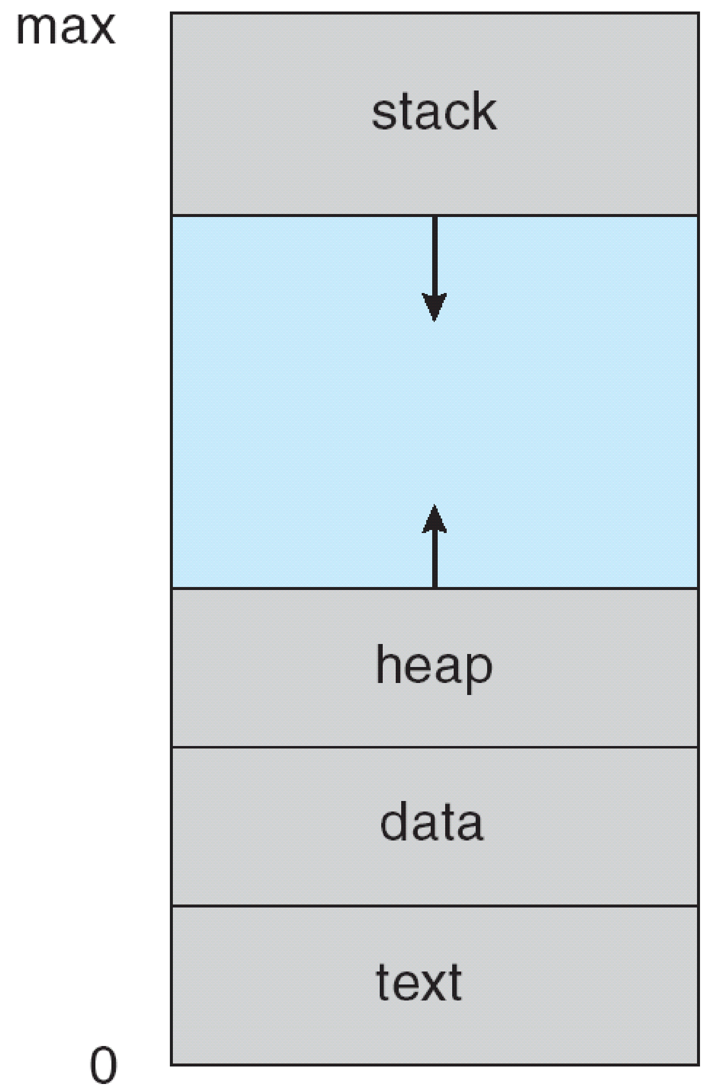
Multi-Programming

- Program
  - Program
  - Job
  - Task
  - …
- Concurrent
  - Single CPU: Pseudo 伪 parallelism
  - Multiprocessor: Parallelism

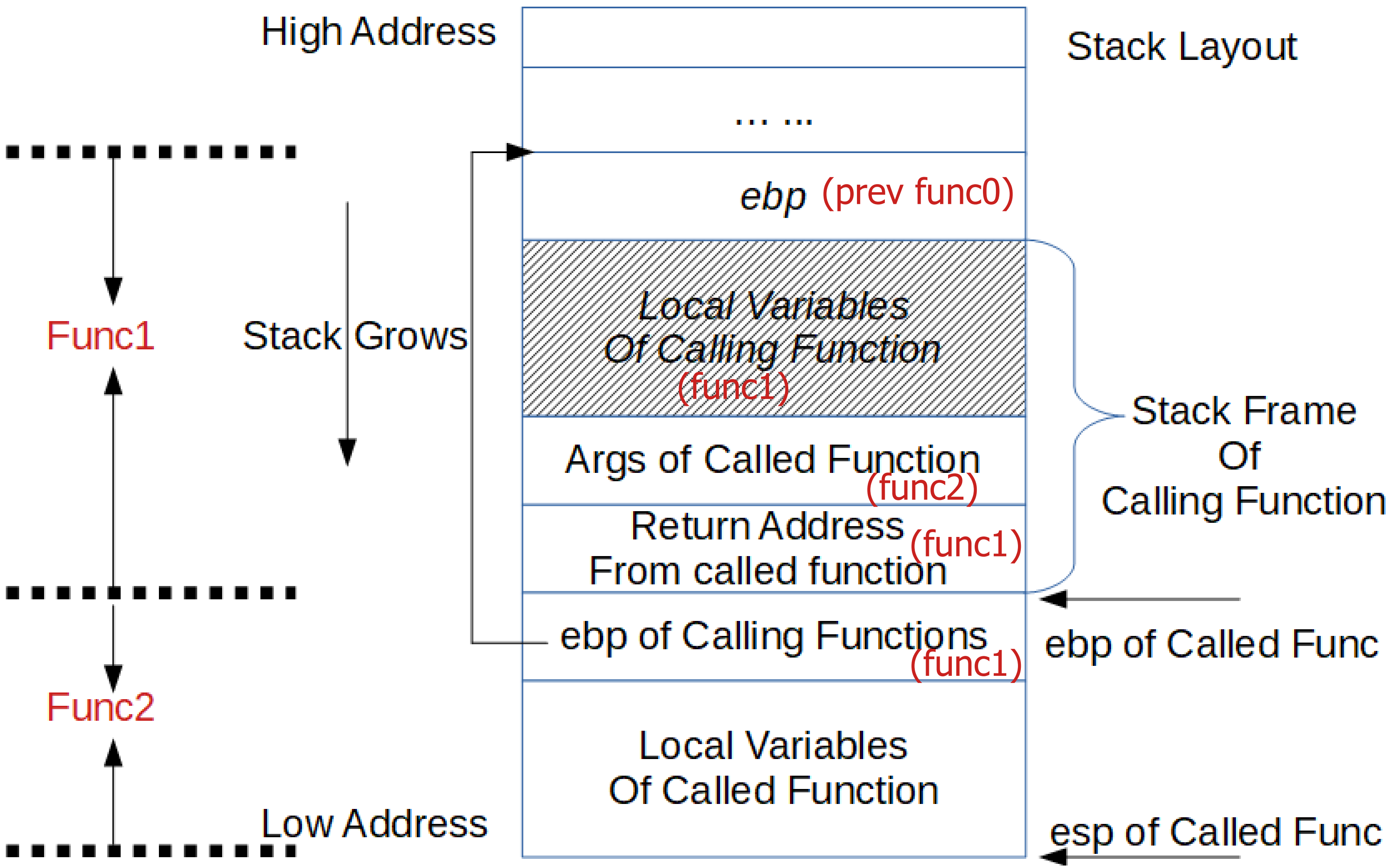
Process

- 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

Process in Memory



Stack Frame of Function



Multi-Processes

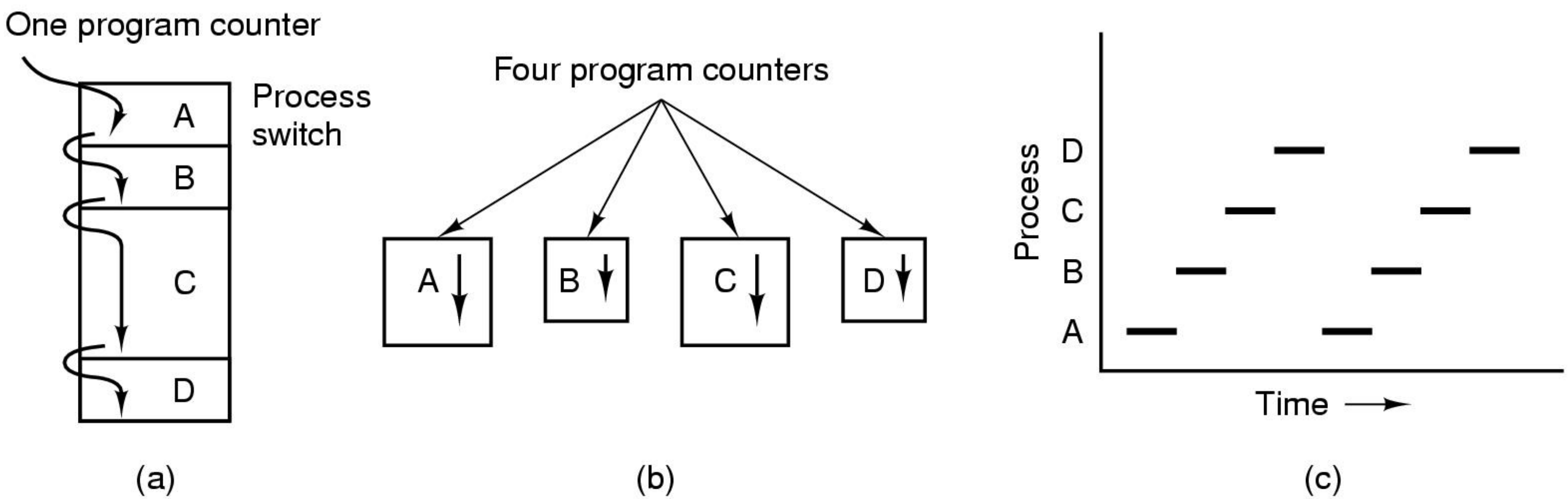


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Program v.s. Process

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

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

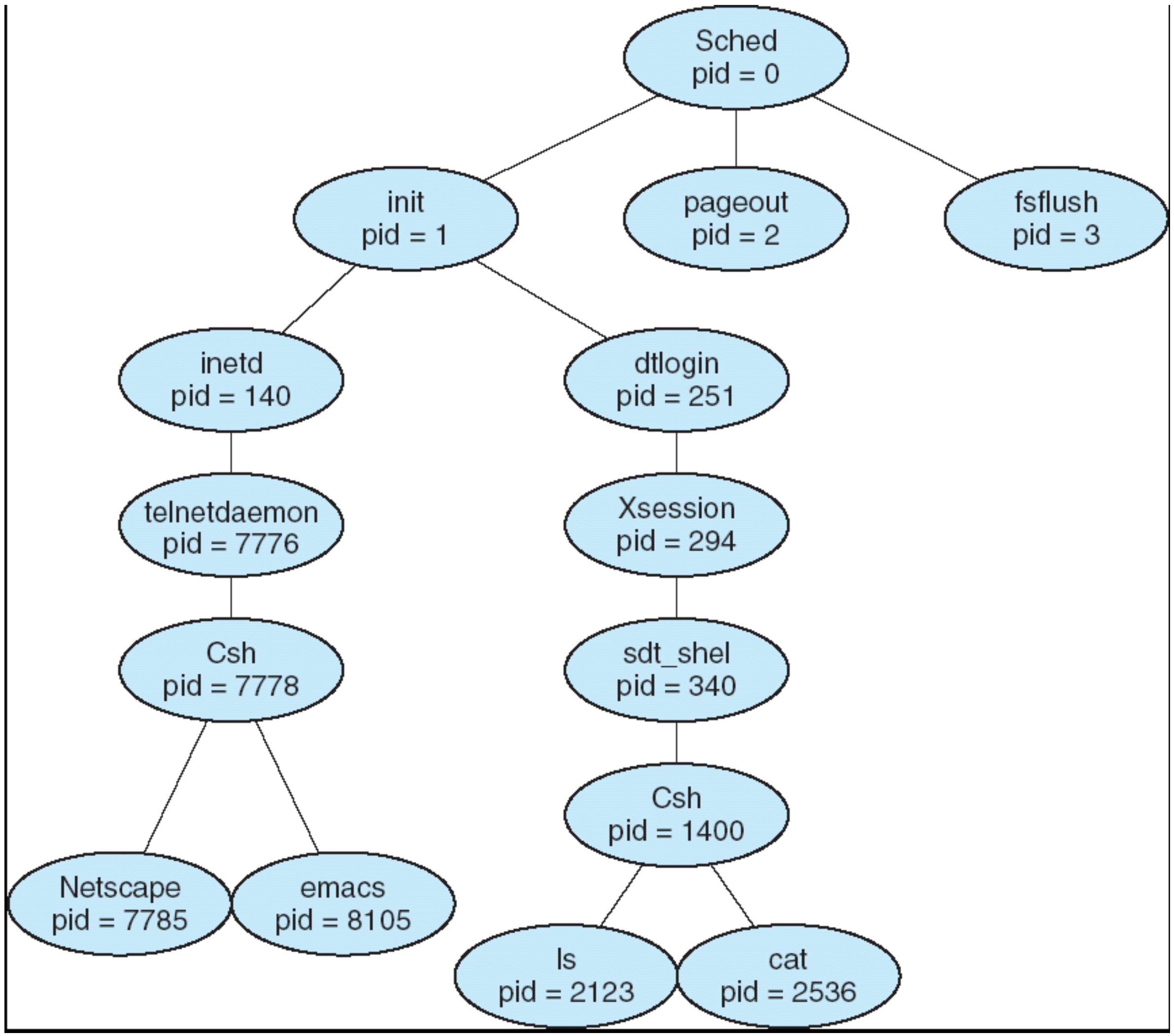
```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
           fprintf(stderr, "Fork Failed");
           exit(-1);
    }
    else if (pid == 0) { /* child process */
           execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent 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)

Process Hierarchies

- UNIX
  - Tree
  - Process group



Process Hierarchies

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

Process States

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

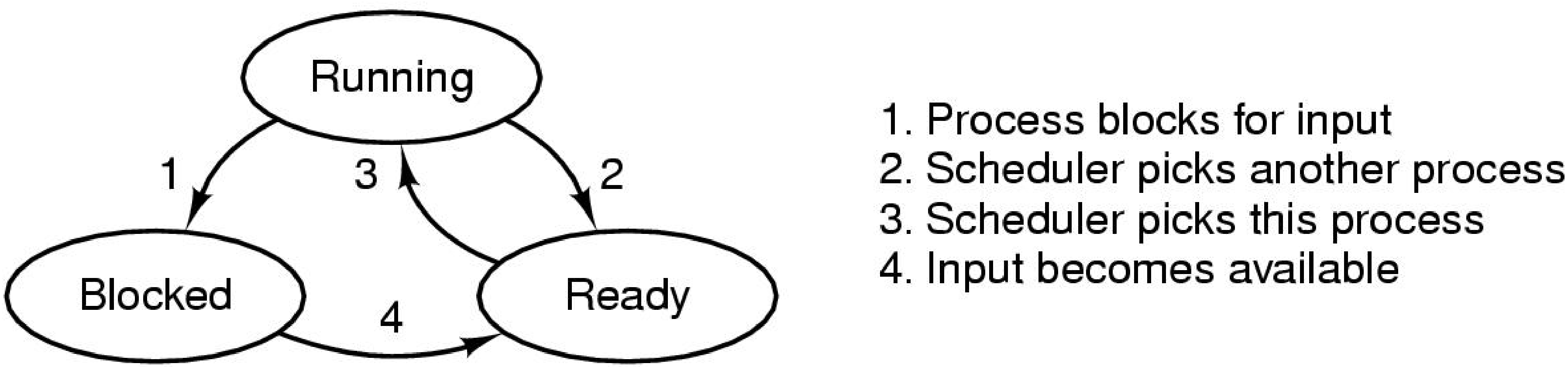
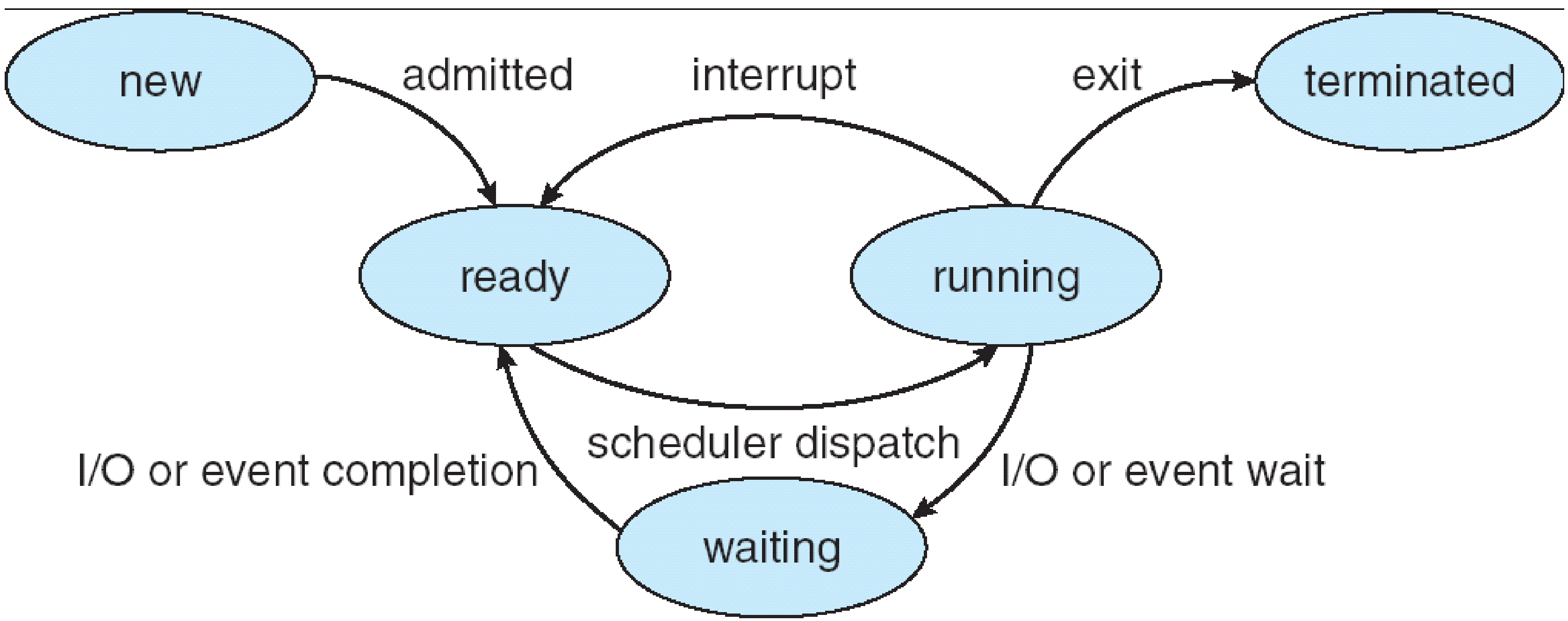


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process States



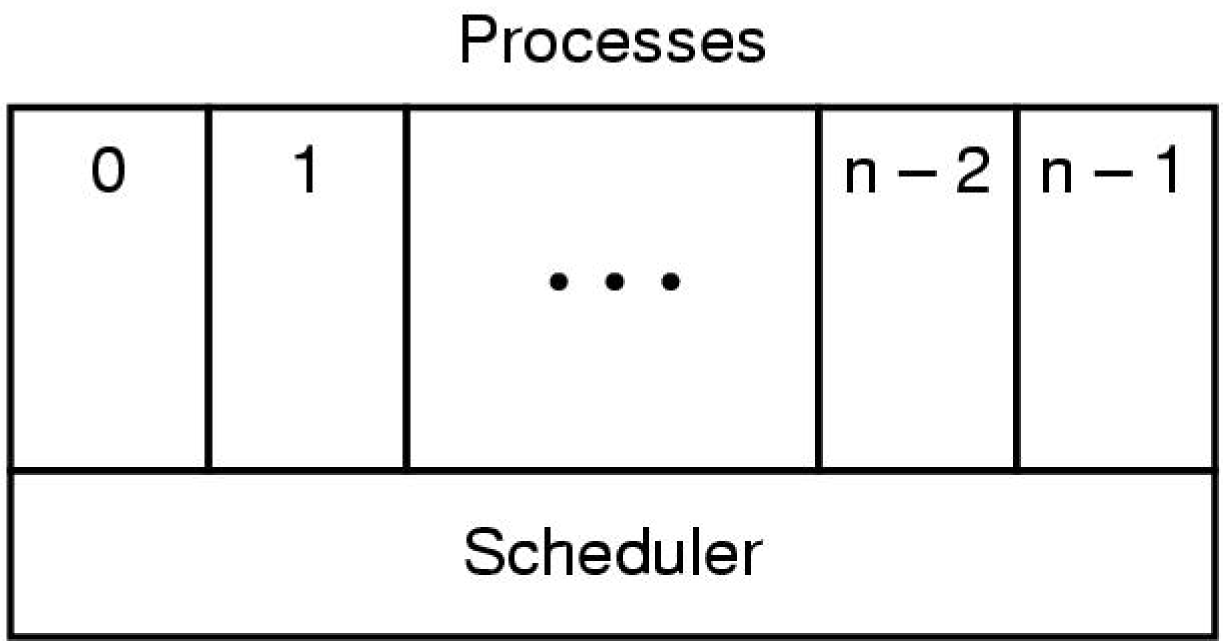
Implementation of Process

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

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
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		

Implementation of Process

- scheduler



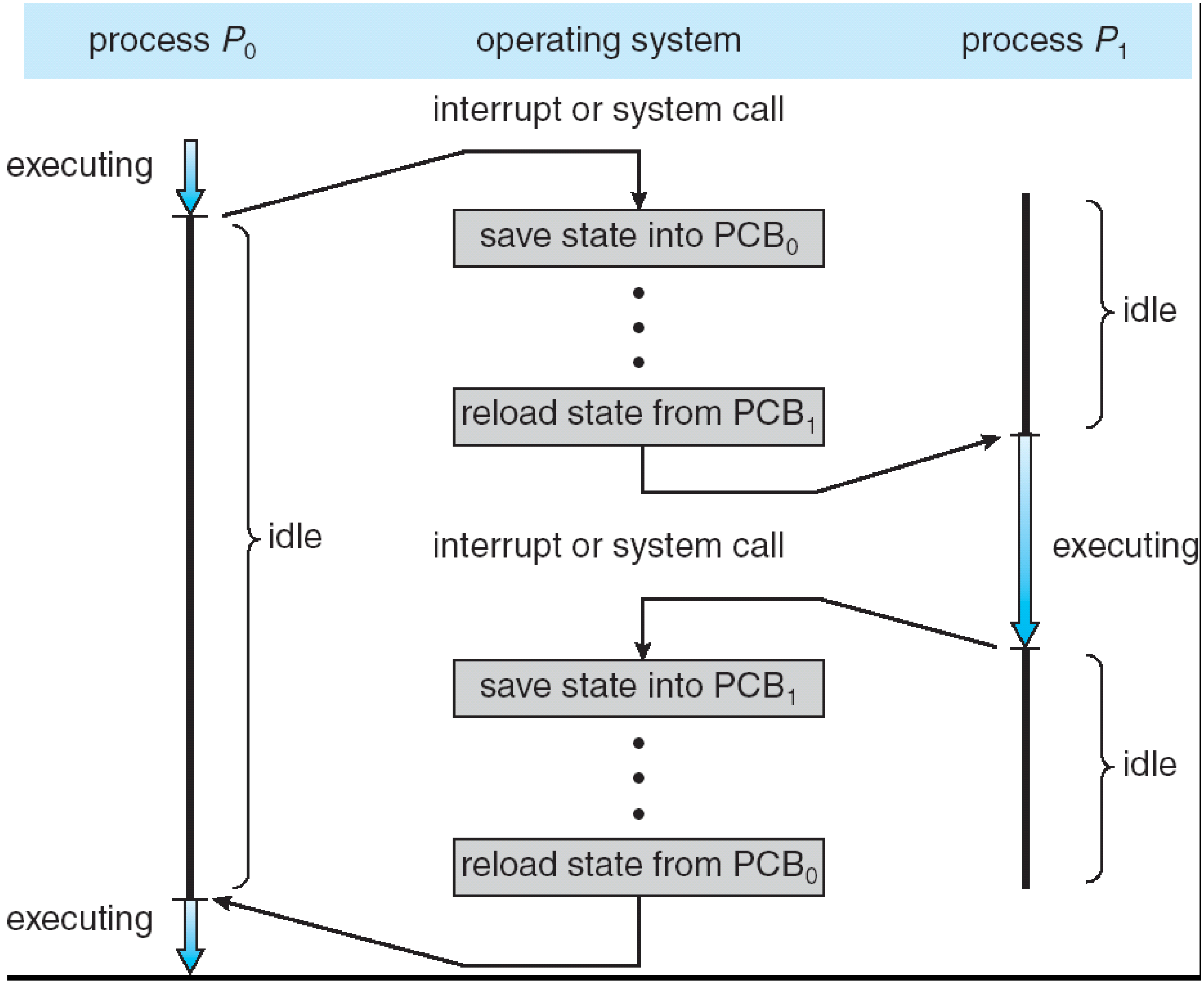
Implementation of Process

- 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.

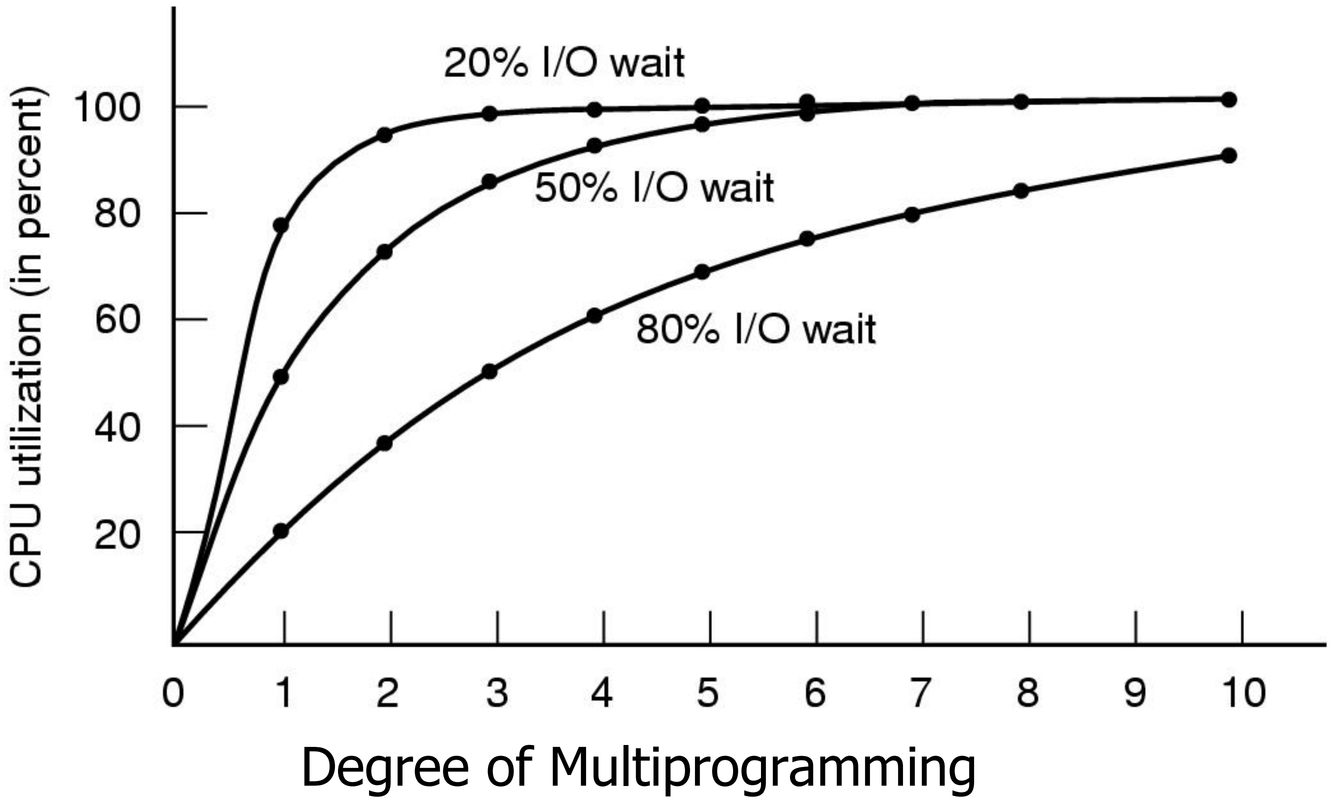
Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

CPU Switch From Process to Process



Modeling Multiprogramming

- CPU utilization
  - $=1 - p^n$
  - P: the time waiting for I/O to complete

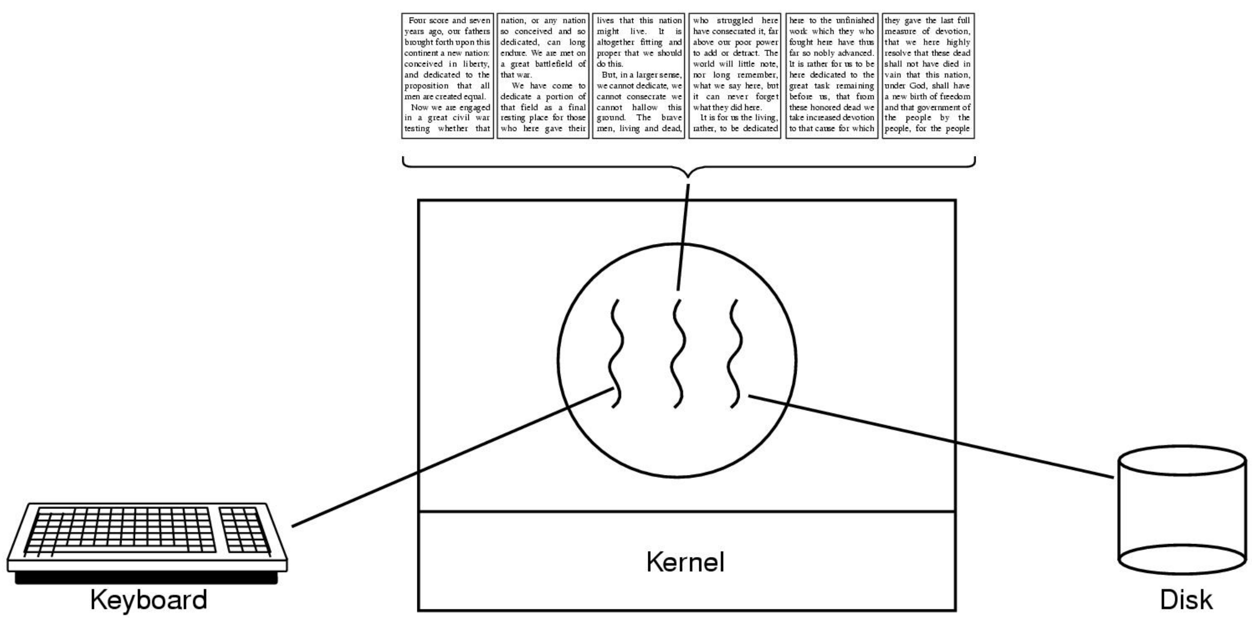


Modeling Multiprogramming

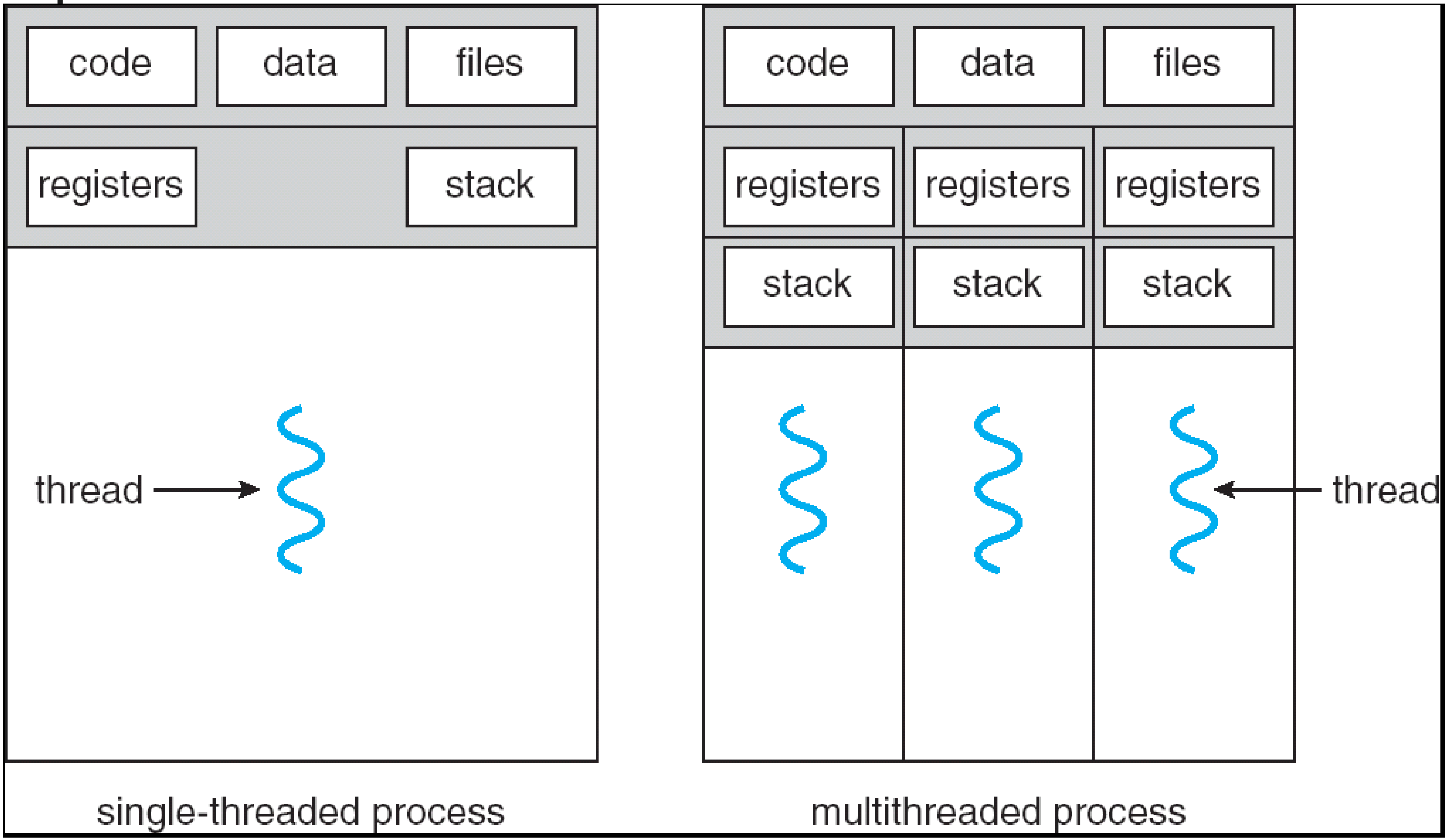
- e.g.
- A computer
  - 512MB memory
  - 128MB for OS, 128MB/Process
  - 80% time to wait I/O
  - CPU utilization=$1-0.8^3$=49%
- Add second memory: 512MB
  - CPU utilization=$1-0.8^7$=79%
- Add third memory: 512MB
  - CPU utilization=?

Thread

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



Single and Multithreaded Processes



Bebefits of MultiThreads

- Multi-threaded programming
  - Responsiveness
  - Resource sharing
  - Economy
  - Utilization of multiprocessor architecture

Cases of MultiThreads

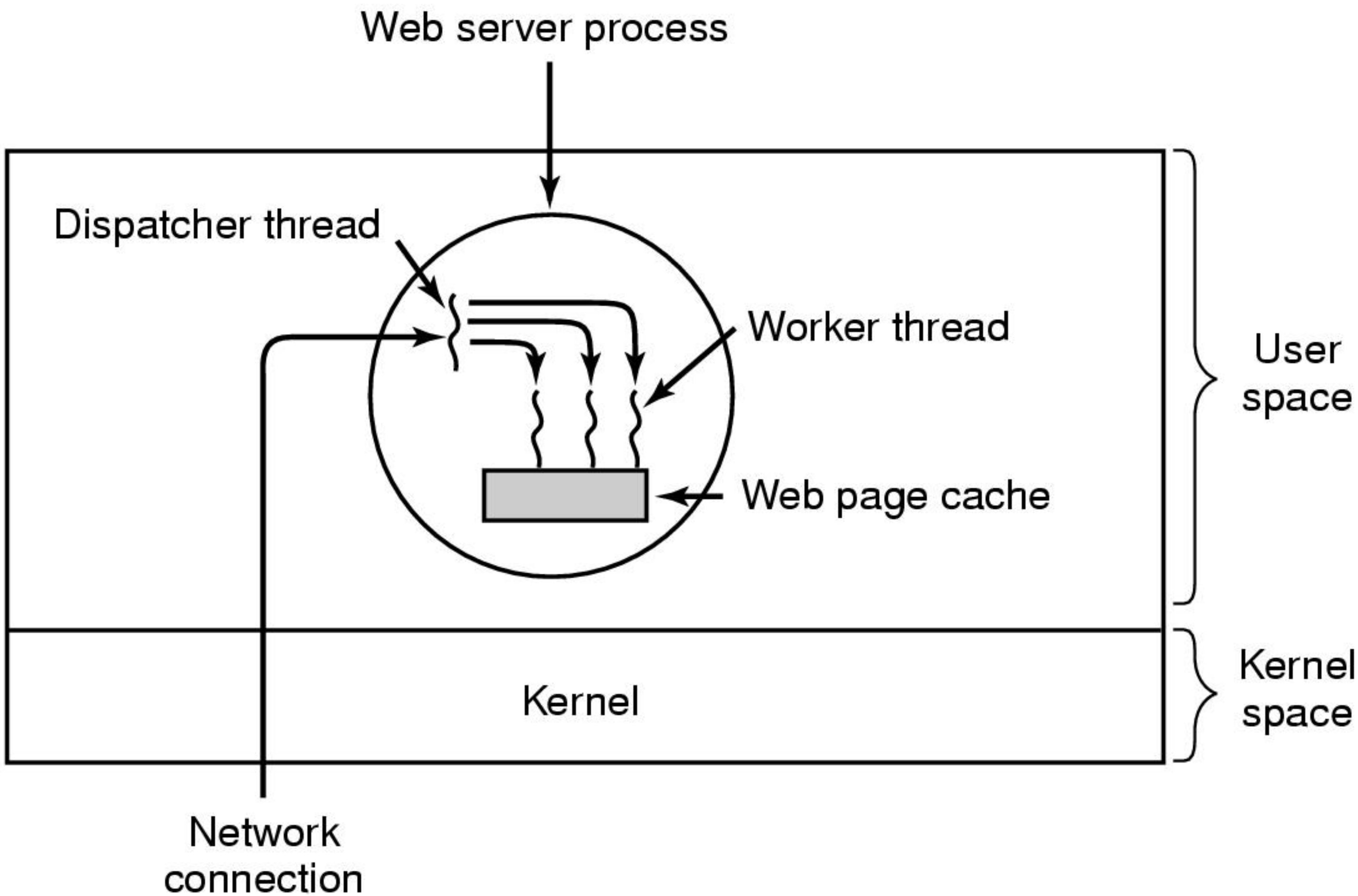


Figure 2-8. A multithreaded Web server.

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)

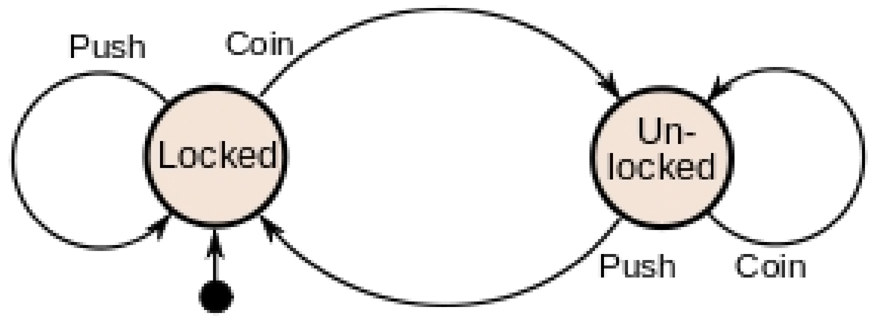
Figure 2-9. A rough outline of the code for Fig. 2-8.
(a) Dispatcher thread. (b) Worker thread.

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

Finite-State Machine (FSM)

- e.g. a turnstile



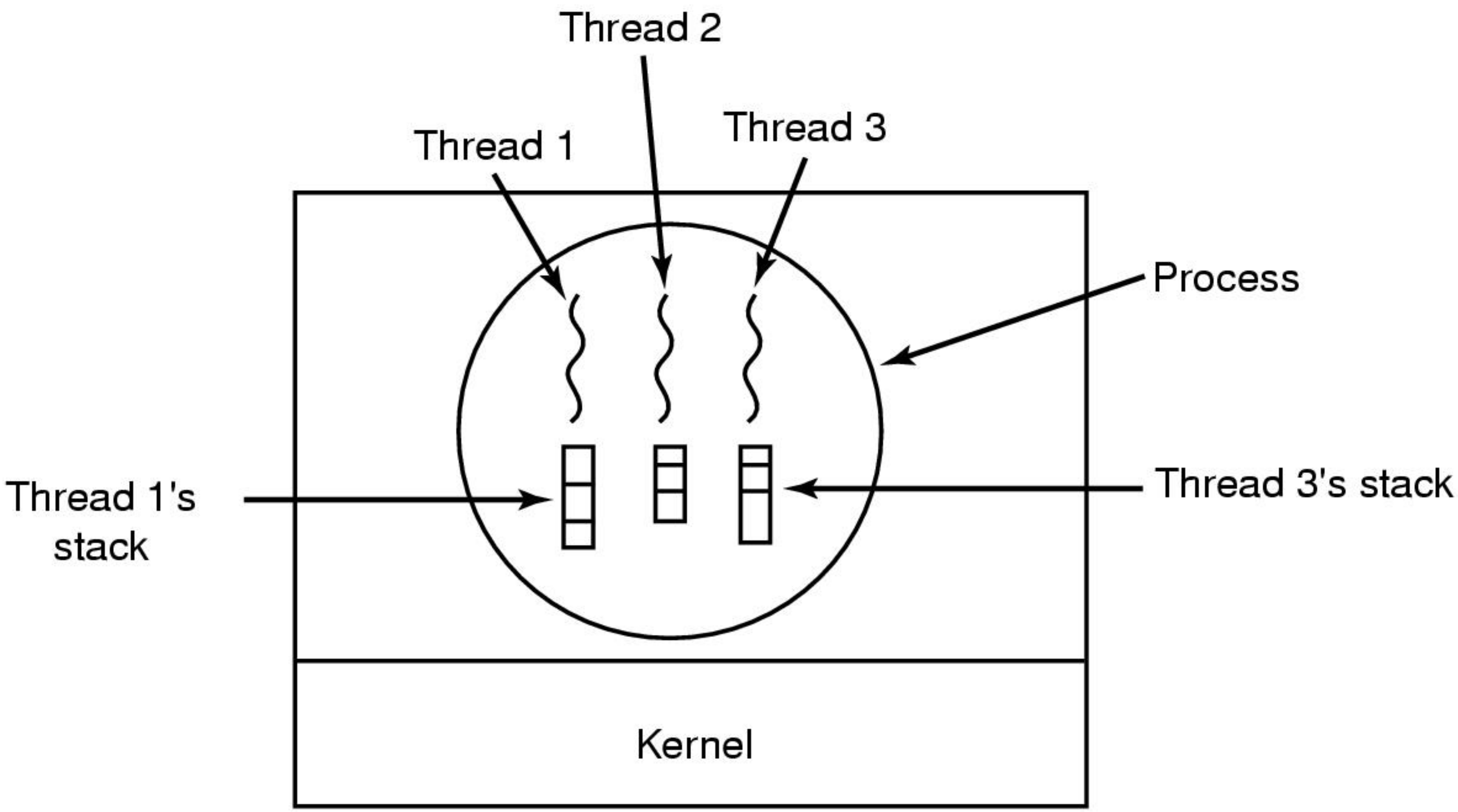
The Classical Thread Model

Private Data of Thread

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.

Each thread has its own stack



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 1: 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 %d0, 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 %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Example 2a: POSIX Threads

```
#include <pthread.h>
typedef struct ST_ThreadArgs{
  int m_index;
  char *m_path;
  pthread_t m_id;
} THREADARGS, *PTHREADARGS;

void *  my_thread (void *aArgs){
  PTHREADARGS mArgs = (PTHREADARGS) aArgs;
  if (mArgs!=NULL){
mArgs->m_id=pthread_self();
printf ("In thread: hello the %dth thread(%ld), The value of m_path is %s\n",
  mArgs->m_index, mArgs->m_id, mArgs->m_path);
  }else{
  fprintf(stderr,"In thread(%ld): args is null\n",pthread_self());
  }
  pthread_exit (NULL);
}
```

Example 2b: POSIX Threads

```
int main (int argc, char *argv[]){
  int i;
  pthread_t *threadsArray;
  THREADARGS *argsArray;
  int status;
  void* res=NULL;
  printf ("Parent %d: begin\n", getpid ());
  printf ("arguments list of %s:\n", argv[0]);
  for (i = 0; i < argc; i++)
    {
      printf ("%3d %s\n", i, argv[i]);
    }
  if (argc <= 1)    return 0;
  threadsArray = (pthread_t *) malloc (sizeof (pthread_t) * (argc - 1));
  argsArray = (THREADARGS *) malloc (sizeof (THREADARGS) * (argc - 1));
```

Example 2c: POSIX Threads

```
  for (i = 1; i < argc; i++){
argsArray[i-1].m_index=i-1;
argsArray[i-1].m_path=argv[i];
*(threadsArray+i-1)=-1;
status = pthread_create(threadsArray+i-1, NULL, my_thread, (void*)(argsArray+i-1));
if (status!=0){
fprintf(stderr,"Failed to Create the %dth thread with code %d\n",i-1, status);
break;
}
    }
```

Example 2d: POSIX Threads

```
i=1;
while(i<argc){
status=pthread_join(threadsArray[i-1],&res);
if (status!=0){
fprintf(stderr,"The %dth thread join failed\n", i-1);
}else{
printf("The %dth thread(%ld) returned %ld\n",
i-1, argsArray[i-1].m_id, (long)status);
}
i++;
}
```

Example 2e: POSIX Threads

```
  free (threadsArray);
  free (argsArray);

  printf ("Parent %d: exited\n", getpid ());
  return EXIT_SUCCESS;
}
```

- Compiling:

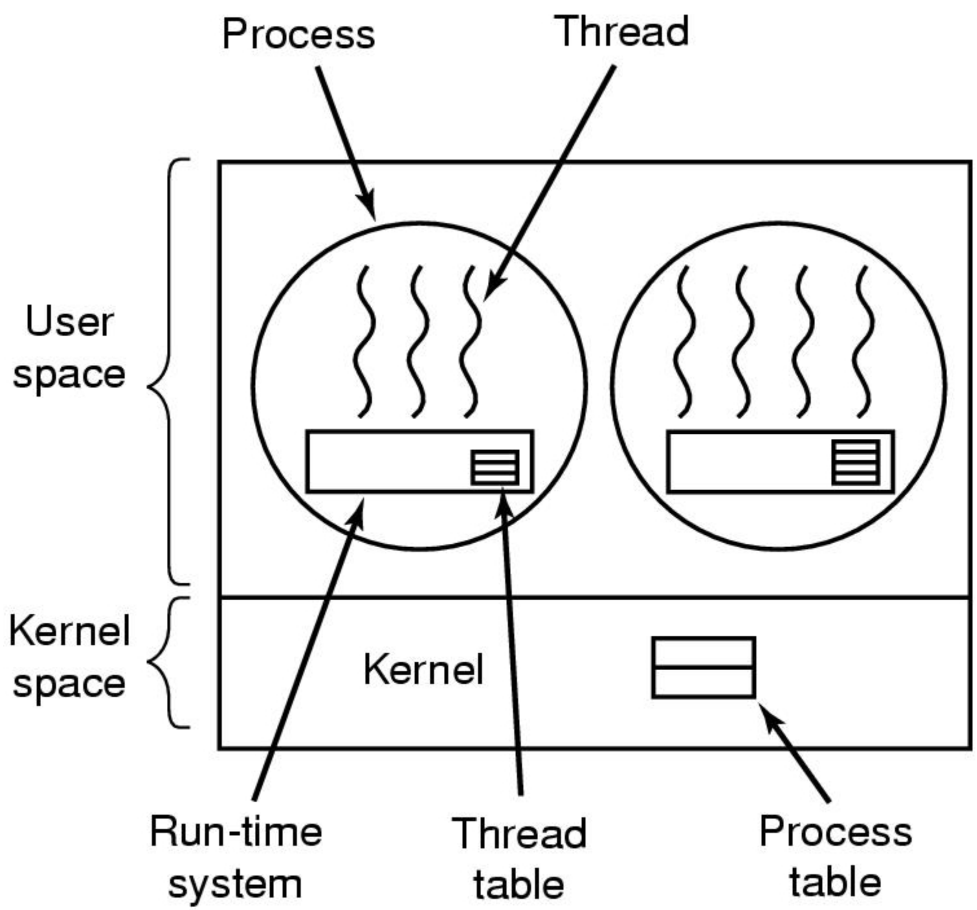
gcc -o multithread multithread.c -lpthread

Example 2f: POSIX Threads

```
$ ./multithread /home /usr abc
Parent 26643: begin
arguments list of ./multithread:
  0 ./multithread
  1 /home
  2 /usr
  3 abc
In thread: hello the 2th thread(140133858793024), The value of m_path is abc
In thread: hello the 0th thread(140133875578432), The value of m_path is
/home
In thread: hello the 1th thread(140133867185728), The value of m_path is /usr
The 0th thread(140133875578432) returned 0
The 1th thread(140133867185728) returned 0
The 2th thread(140133858793024) returned 0
Parent 26643: exited
$
```

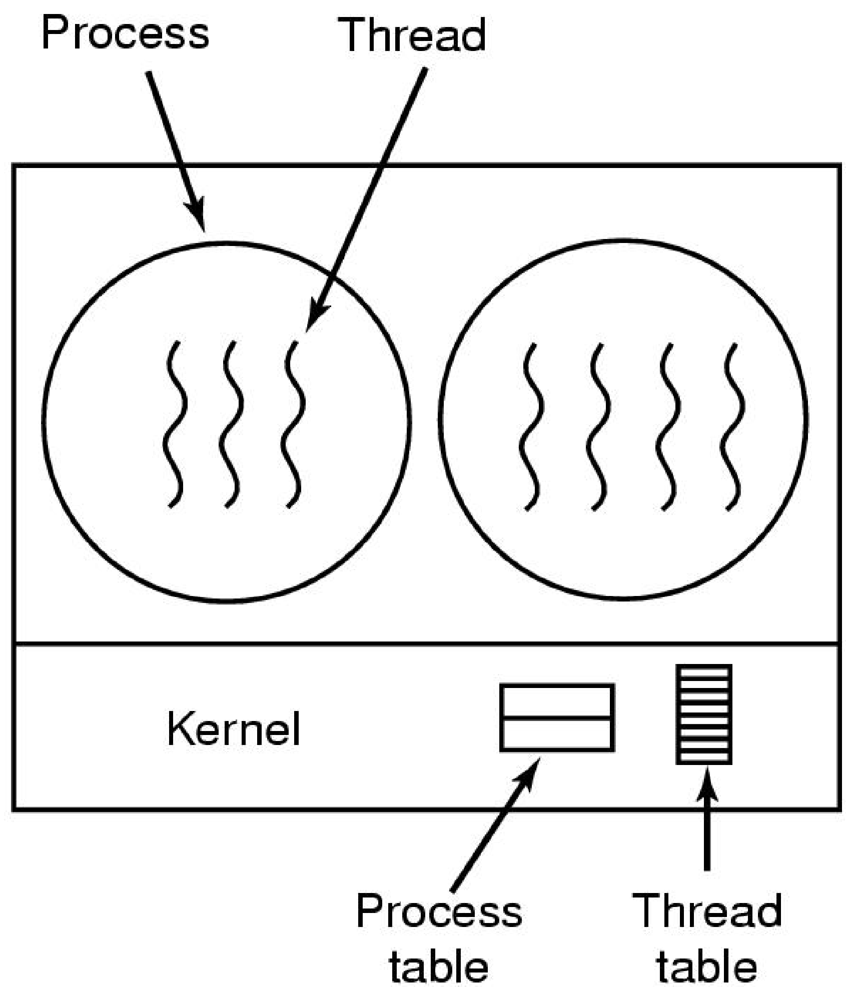
Different types of threads

- User-Level Thread
  - In user space



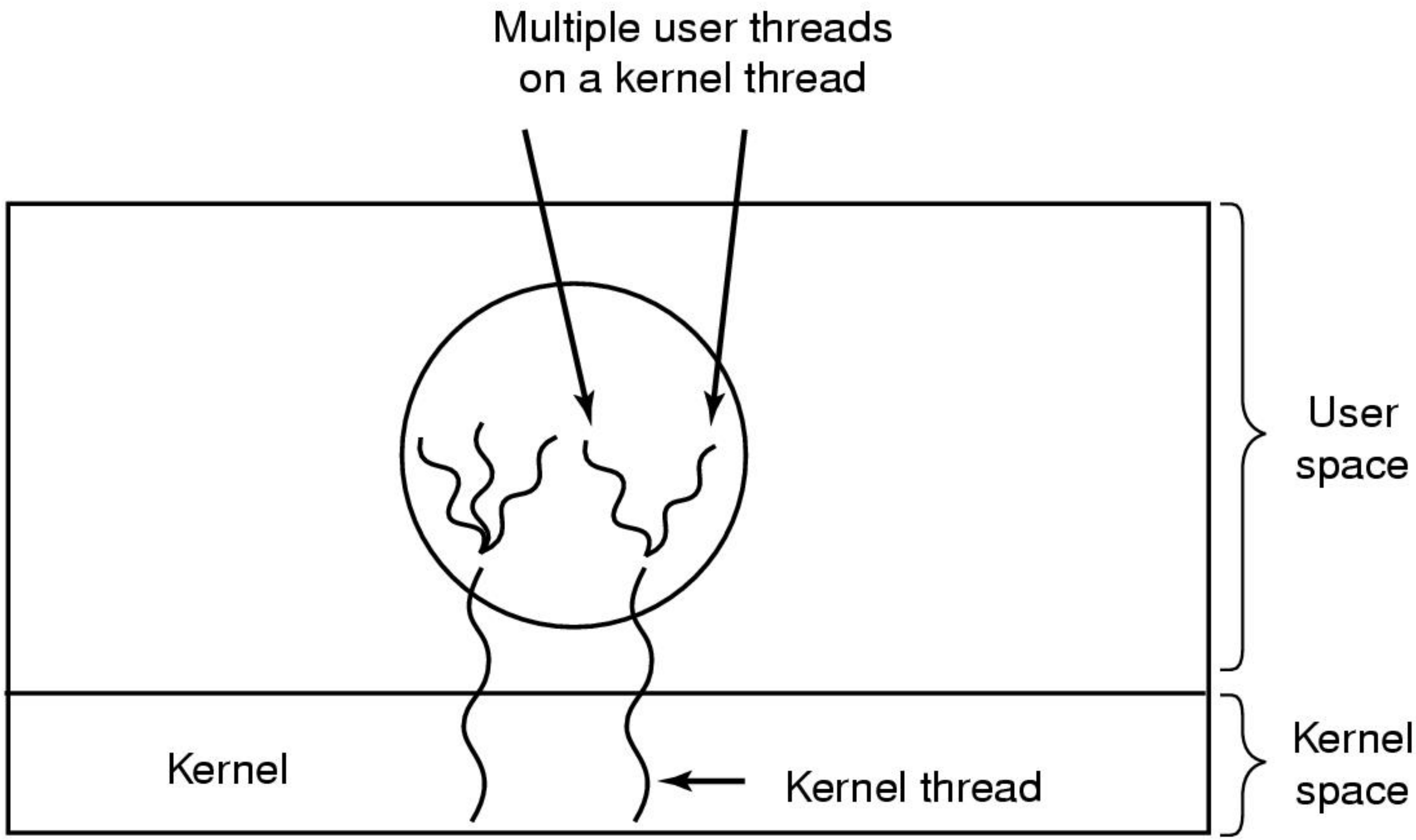
Different types of threads

- Kernel-Level Thread
  - In os kernel

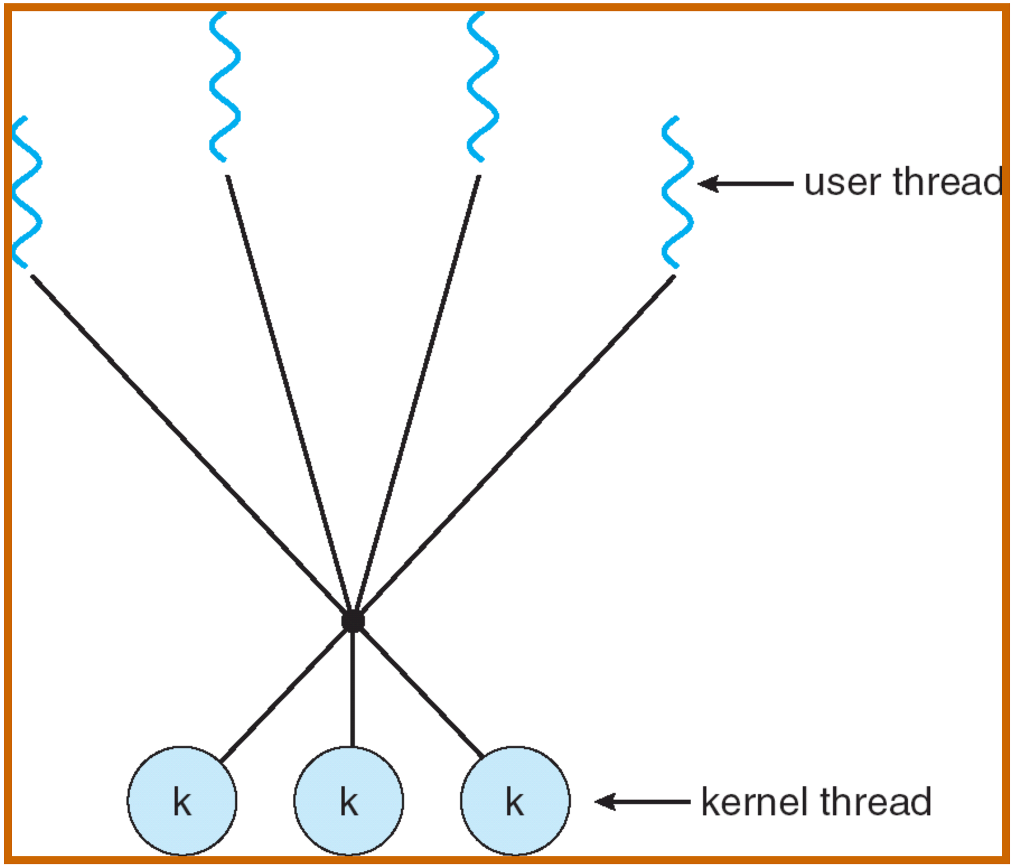


Different types of threads

- Hybrid Implementation Thread
  - Both kernel and User Thread



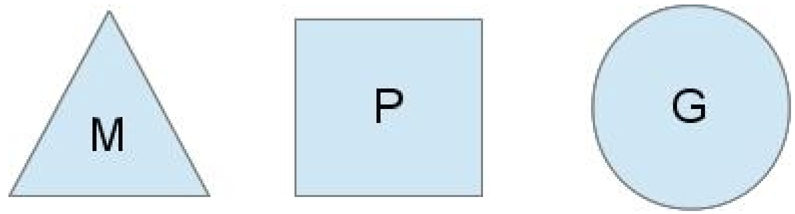
Hybrid Implementation Thread I



M:N (M:1, 1:1, M:N)

Hybrid Implementation Thread II

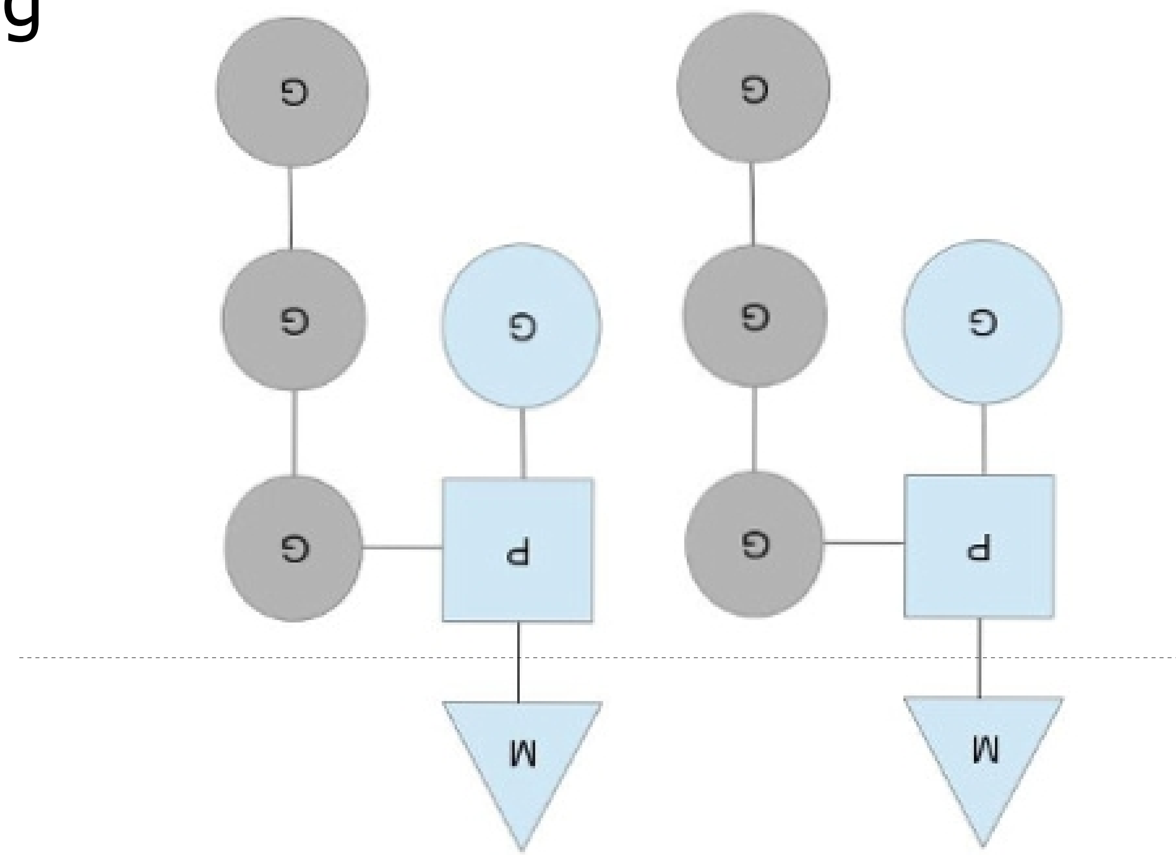
- coroutine 协程 (M:N)
  - Golang



  - 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 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.")
}
```

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

```
Hello, 世界
B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_10 B_11 B_12 A_1 A_2 A_3 A_4 A_5 A_6 B_13 B_14 B_15 B_16
B_17 B_18 B_19 B_20 A_7 A_8 A_9 A_10 A_11 A_12 A_13 A_14 A_15 A_16 A_17 A_18 A_19 A_20
End.
```

Threading Issues

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

Semantics of fork() and exec()

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

Thread cancellation

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

Signal 信号 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

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

Pop-Up Threads

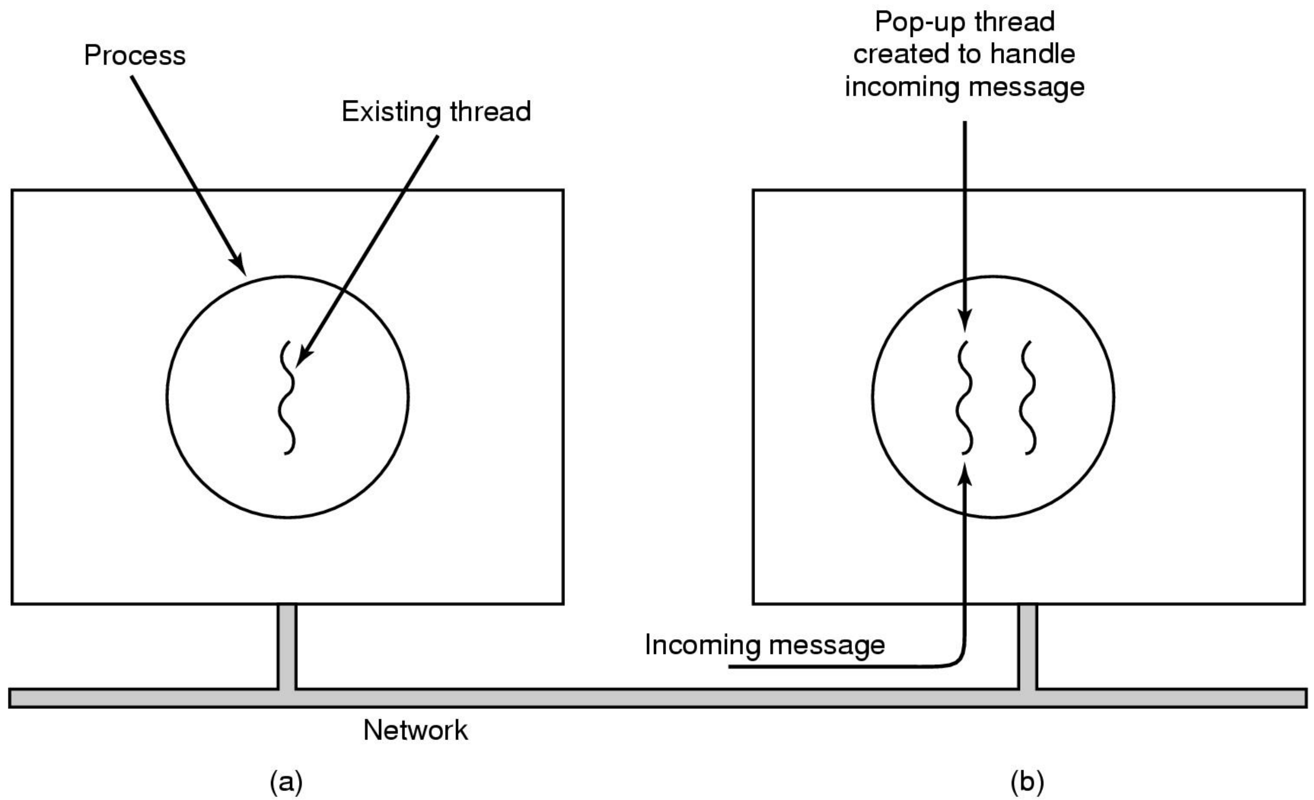


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

TSD: 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)

Making Single-Threaded Code Multithreaded

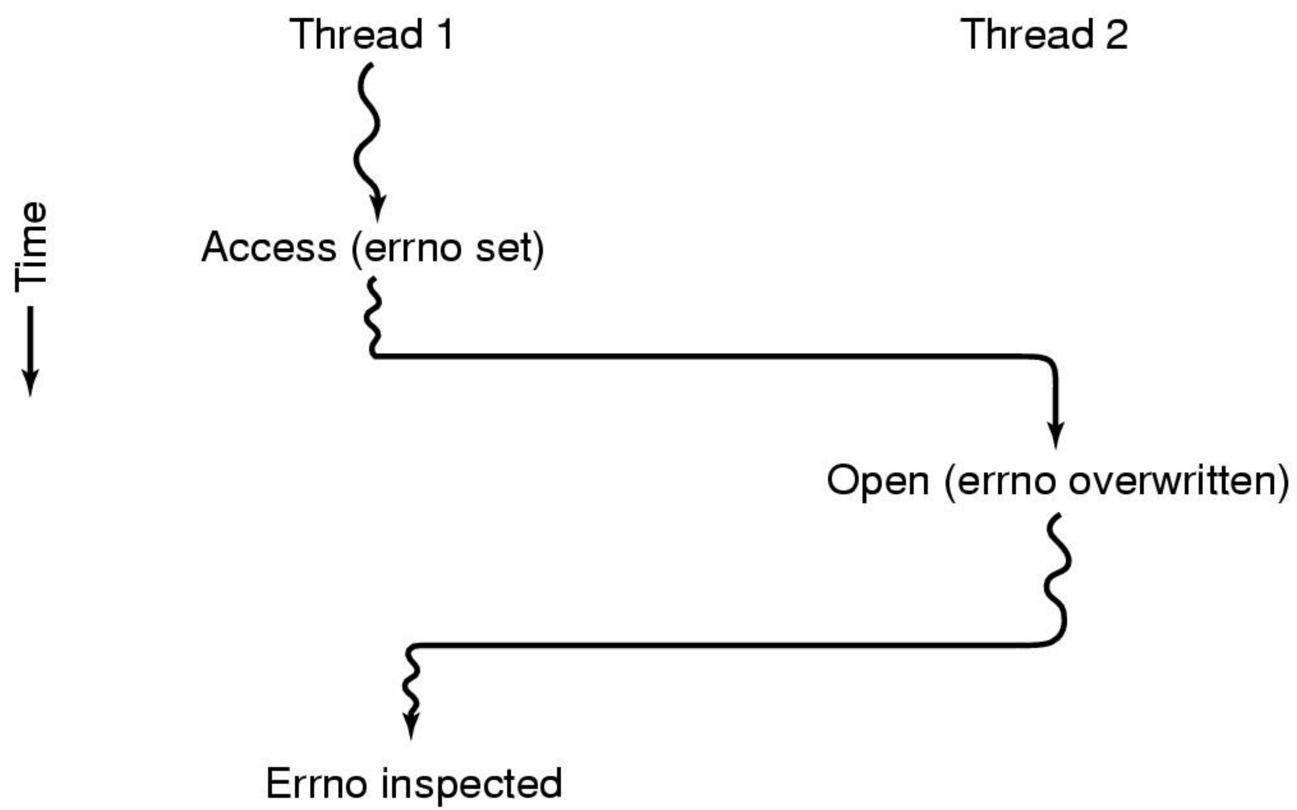


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded

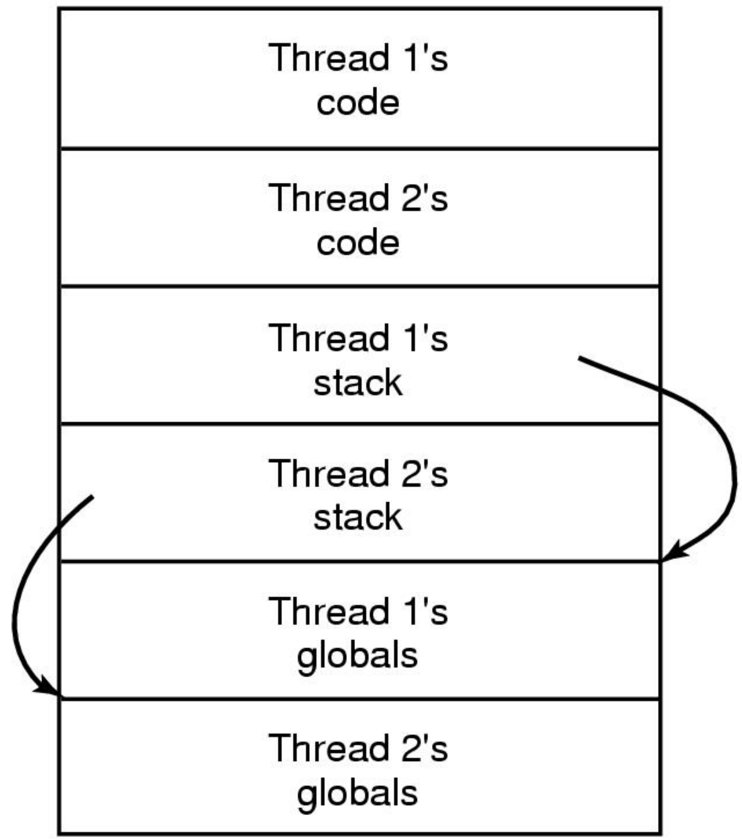


Figure 2-20. Threads can have private global variables.

Thread Scheduler Activations 调度激活

- Goal
  - mimic 模仿 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 上行
  - Virtual processors
  - Run-time system
  - ? layer n may not call procedures in layer n + 1

Summary

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

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

рахмат
danke
謝謝
ngiyabonga
teşekkür ederim
Баярлалаа
mersi
kia ora
barka
welalin
spas
tack
mahalo
tapadh leat
спасибо
faafetai lava
vinaka
спасибі
blagodaram
dank je
misaotra
matondo
paldies
grazzi
kiitos
dankie
nanni
nandri
dhanyavad
mauruuru
köszönöm
thank you
akun
dankon
aciú
gracias
хвала
asante
manana
obrigada
tenki
bedankt
bayarlalaa
gràcie
hvala
enkosi
djiere dieuf
tau
mochchakkeram
murakoze
chokrane
mamnun
дякую
dziękuję
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
sobodi
dekuji
obrigado
sagolun
sukriya
kop khun krap
taiku
arigatô
takk
dakujem
trugarez
mèsi
didi madloba
najis tuke
ありがとう
tanemirt
grazie
kam sah hamnida
rahmat
terima kasih
rahmet
ευχαριστώ
diolch
dhanyavadagalu
shukriya
mercé
мерси
감사합니다
xièxie
merci