

Operating Systems

Jinghui Zhong (钟竞辉)

Office: B3-515

Email : jinghuizhong@scut.edu.cn



Experiment Schedule

● Chinese students:

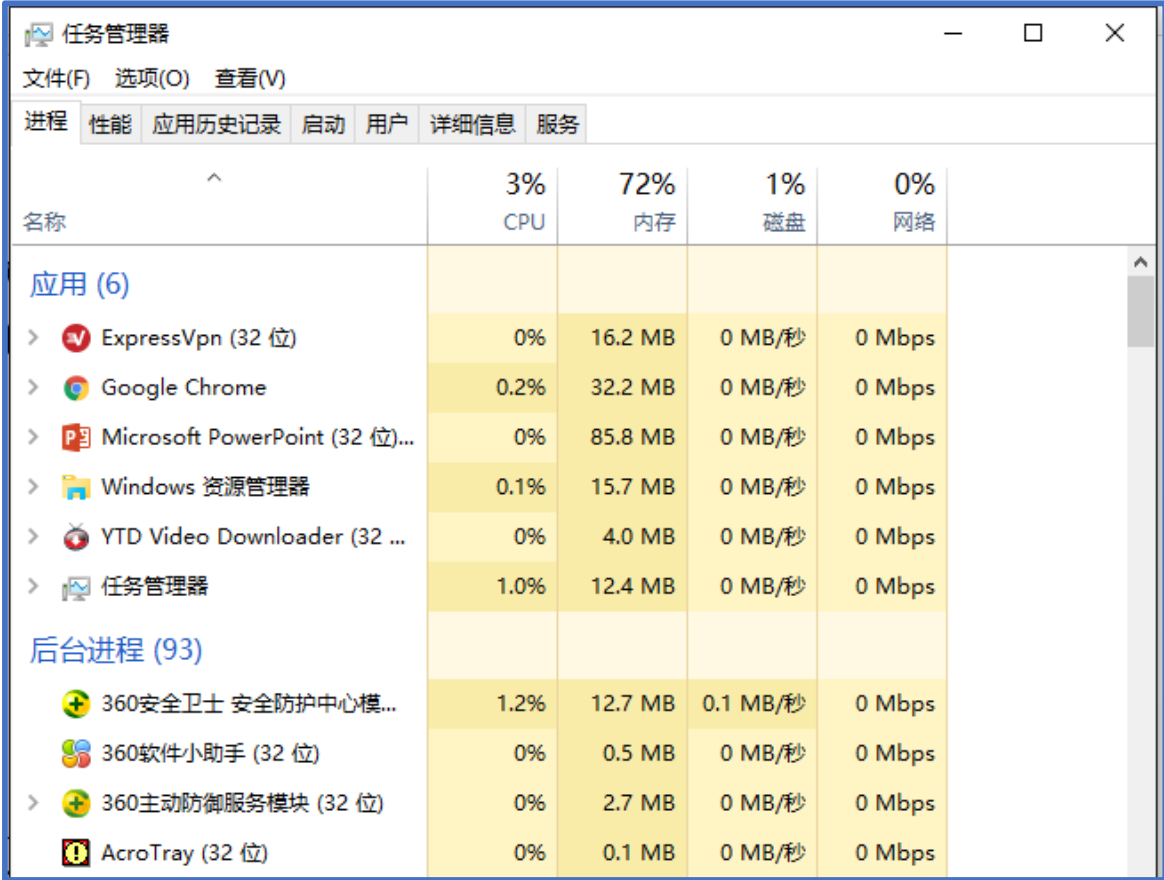
- (1) Sep. 30, 14:00-17:20, B3-138
- (2) Nov. 11, 14:00-17:20, **B3-231**
- (3) Nov. 18, 08:50-12:15, B3-138
- (4) Nov. 20, 08:50-12:15, B3-138

● Oversea students:

- (1) Sep. 30, 14:00-17:20, B3-138
- (2) Nov. 18, 08:50-12:15, B3-138
- (3) Nov. 20, 08:50-12:15, B3-138
- (4) Nov. 22, 08:50-12:15, **B3-234**

Processes

- Multiprogramming: when the system is booted, many processes are running simultaneously, e.g., QQ, WeChat, etc.



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running applications and background processes, along with their CPU, memory, disk, and network usage. The window title is '任务管理器' (Task Manager). The menu bar includes '文件(F)', '选项(O)', and '查看(V)'. The tabs are '进程' (Processes), '性能' (Performance), '应用历史记录' (App history), '启动' (Startup), '用户' (Users), '详细信息' (Details), and '服务' (Services). The table has columns for '名称' (Name), 'CPU', '内存' (Memory), '磁盘' (Disk), and '网络' (Network). The processes are grouped into '应用 (6)' (Applications) and '后台进程 (93)' (Background processes).

名称	3% CPU	72% 内存	1% 磁盘	0% 网络
应用 (6)				
> ExpressVpn (32 位)	0%	16.2 MB	0 MB/秒	0 Mbps
> Google Chrome	0.2%	32.2 MB	0 MB/秒	0 Mbps
> Microsoft PowerPoint (32 位)...	0%	85.8 MB	0 MB/秒	0 Mbps
> Windows 资源管理器	0.1%	15.7 MB	0 MB/秒	0 Mbps
> YTD Video Downloader (32 ...	0%	4.0 MB	0 MB/秒	0 Mbps
> 任务管理器	1.0%	12.4 MB	0 MB/秒	0 Mbps
后台进程 (93)				
+ 360安全卫士 安全防护中心模...	1.2%	12.7 MB	0.1 MB/秒	0 Mbps
+ 360软件小助手 (32 位)	0%	0.5 MB	0 MB/秒	0 Mbps
> + 360主动防御服务模块 (32 位)	0%	2.7 MB	0 MB/秒	0 Mbps
+ AcroTray (32 位)	0%	0.1 MB	0 MB/秒	0 Mbps

Multiprogramming

- Read the first two paragraphs of section 2.1.1 (pp.86), and find **a concise sentence** to describe multiprogramming.

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just **processes** for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming**, as we saw in Chap. 1.

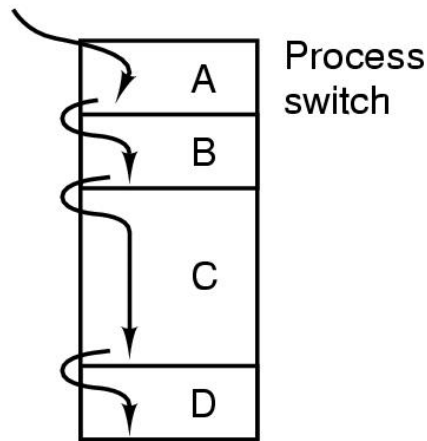
In Fig. 2-1(a) we see a computer multiprogramming four programs in memory. In Fig. 2-1(b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory. In Fig. 2-1(c) we see that, viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.



Multiprogramming

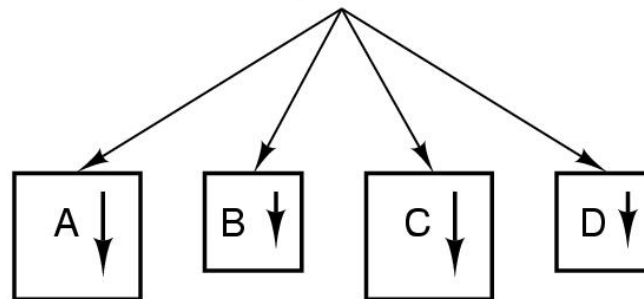
- The CPU switches from process to process quickly, running each for tens or hundreds of milliseconds.
- At anytime, the CPU is running only one process.

One program counter

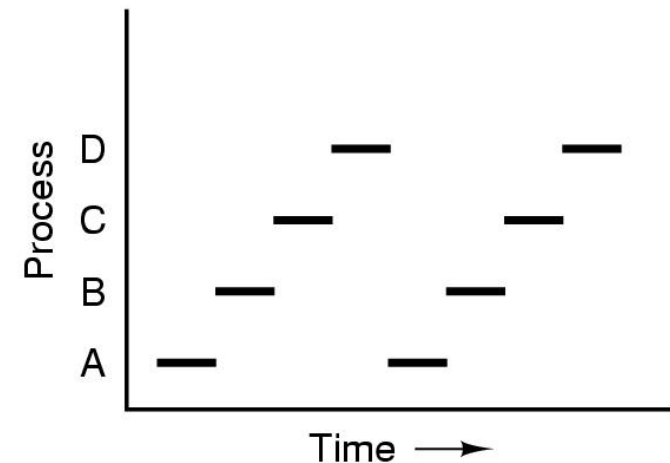


(a)

Four program counters



(b)



(c)

Process Creation

●Events that cause process creation

1. System initialization
2. Created by a running process.
3. User request to create a new process
4. Initiation of a batch job

●**Foreground processes:** processes that interact with users and perform work for them.

●Background processes that handle some incoming request are called **daemons**.

How to write a program to execute another program?



Unix/Linux: Process Creation

- Use `fork()` system call to create a new process. The `execve()` system call can be used to load a new program.

```
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
int main(){
    int i, flag,pid,status;
    for(i = 1; i<=3; i++){
        if((flag=fork())==0){
            printf("In child %d.\n", i);
        }else{
            wait(&status);
        }
    }
    return 1;
}
```



Windows: Process Creation

- Use **CreateProcess** to handles both creation and loading the correct program into the new process.

```
#include<stdio.h>
#include<windows.h>
int main(int argc,char*argv[])
{
    Char szCommandLine[]="notepad";
    STARTUPINFO si={sizeof(si)};
    PROCESS_INFORMATION pi;
    si.dwFlags=STARTF_USESHOWWINDOW;
    si.wShowWindow=TRUE;
    BOOL bRet=CreateProcess( NULL, szCommandLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
    if(bRet)
    {
        CloseHandle(pi.hThread);
        CloseHandle(pi.hProcess);
        printf("New Process's ID: %d\n",pi.dwProcessId);
        printf("New Process's main thread's ID: %d\n",pi.dwThreadId);
    }
    return 0;
}
```



Some Related Links

<http://www.cnblogs.com/hicjiajia/archive/2011/01/20/1940154.html>

http://blog.csdn.net/fisher_jiang/article/details/5608399

<http://blog.csdn.net/bzhxuexi/article/details/23950701>

<https://www.youtube.com/watch?v=xHu7qI1gDPA&list=PLegzNeGav1mcwCib09iz4ioSGDjPB34Ff>

<https://www.youtube.com/watch?v=xVSPv-9x3gk>



Process Termination

● Conditions that cause process termination

- ✓ **Normal exit (voluntary)**

“Exit” in UNIX and “ExitProcess” in Windows.

- ✓ **Error exit (voluntary)**

Example: input file is not exist.

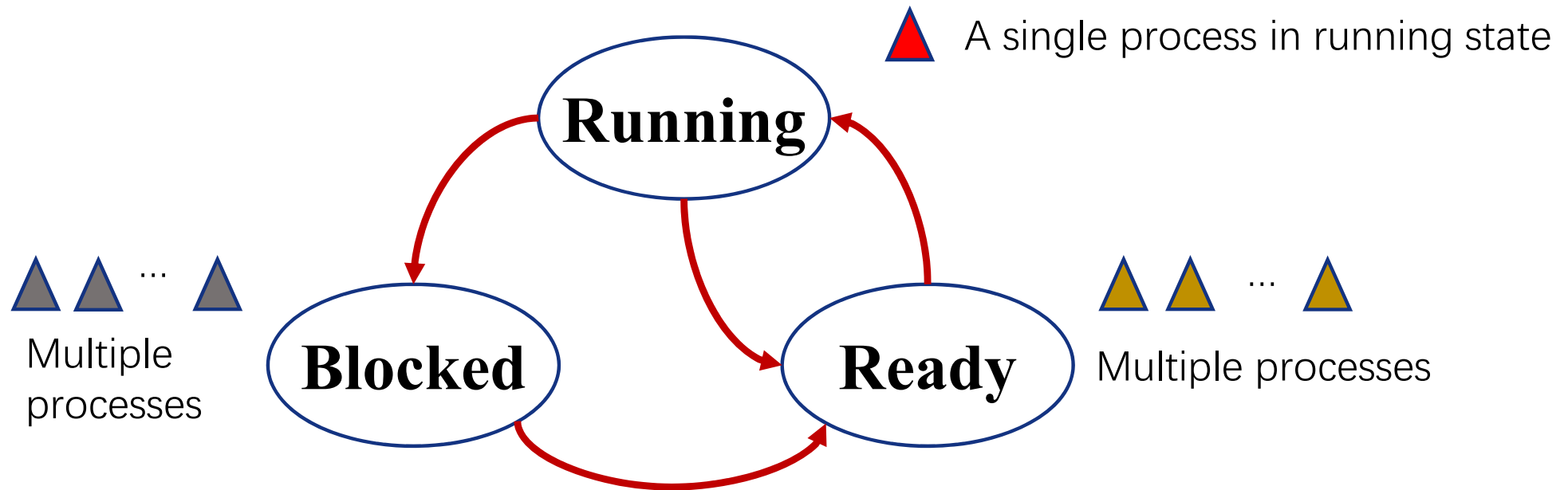
- ✓ **Fatal error (involuntary)**

Example: referencing nonexistent memory.

- ✓ **Killed by another process (involuntary)**

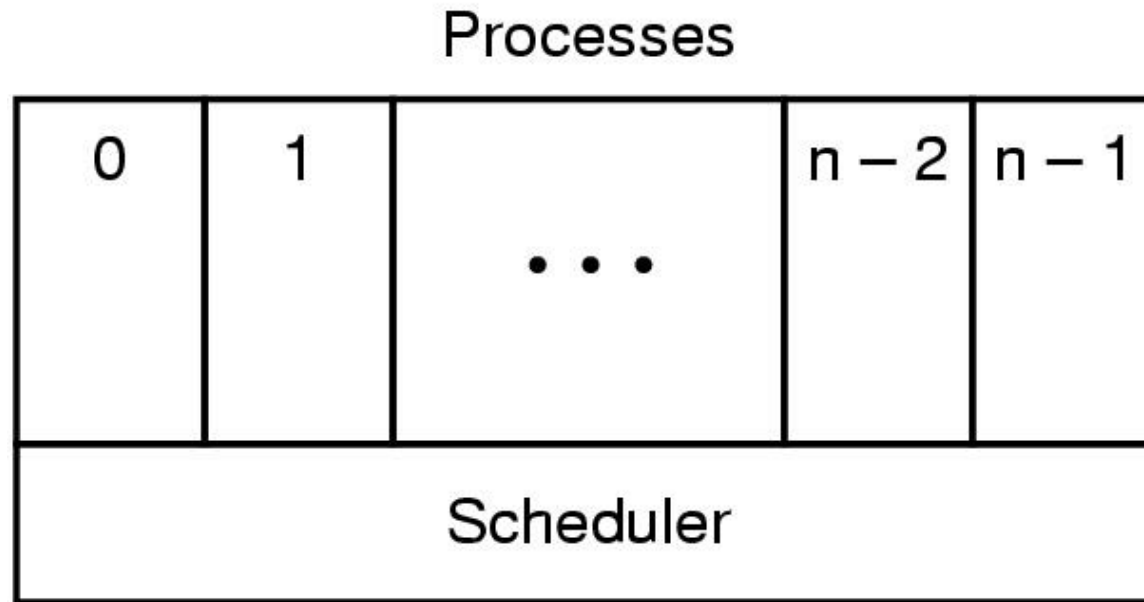
“Kill” in UNIX and “TerminateProcess” in Windows.

Process States



- **Running:** using the CPU at that instant.
- **Ready:** runnable; temporarily stopped to let another process run.
- **Blocked:** unable to run until some external event happens.

Process States



- OS : the lowest layer, handles interrupts, scheduling
- Above that layer are sequential processes
e.g., User processes, terminal processes

Implementation of Processes

- The OS maintains a **Process Table** with one entry (called a **process control block (PCB)**) for each process.

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 Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Implementation of Processes

- When a context switch occurs between processes P1 and P2, the current state of the **RUNNING** process, say P1, is saved in the PCB for process P1 and the state of a **READY** process, say P2, is restored from the PCB for process P2 to the CPU registers, etc. Then, process P2 begins **RUNNING**.
- **Pseudo- parallelism** : the rapid switching between processes gives the illusion of true parallelism and is called pseudo-parallelism.

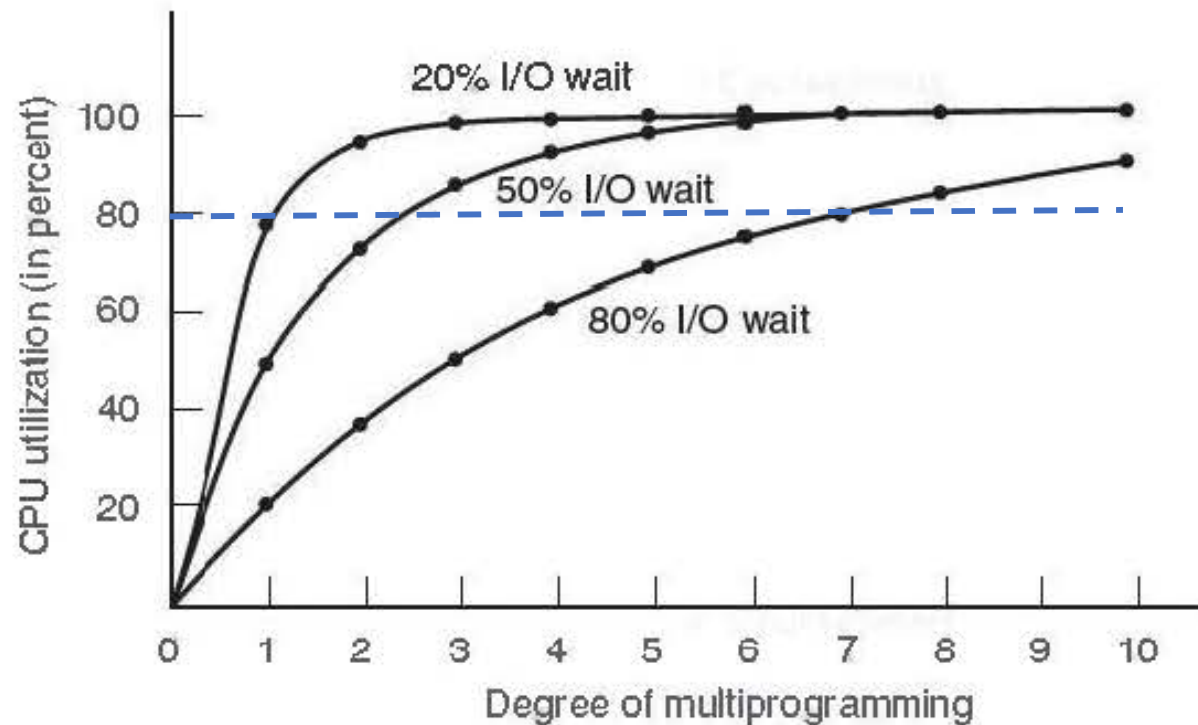
Modeling Multiprogramming

- Suppose a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once. The CPU utilization can be obtained by:

$$\text{CPU utilization} = 1 - p^n$$

CPU Utilization in Multiprogramming

- CPU utilization = $1 - p^n$



CPU utilization as a function of the number of processes in memory.

Thread

● What is a thread?

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Thread

● What is a thread?

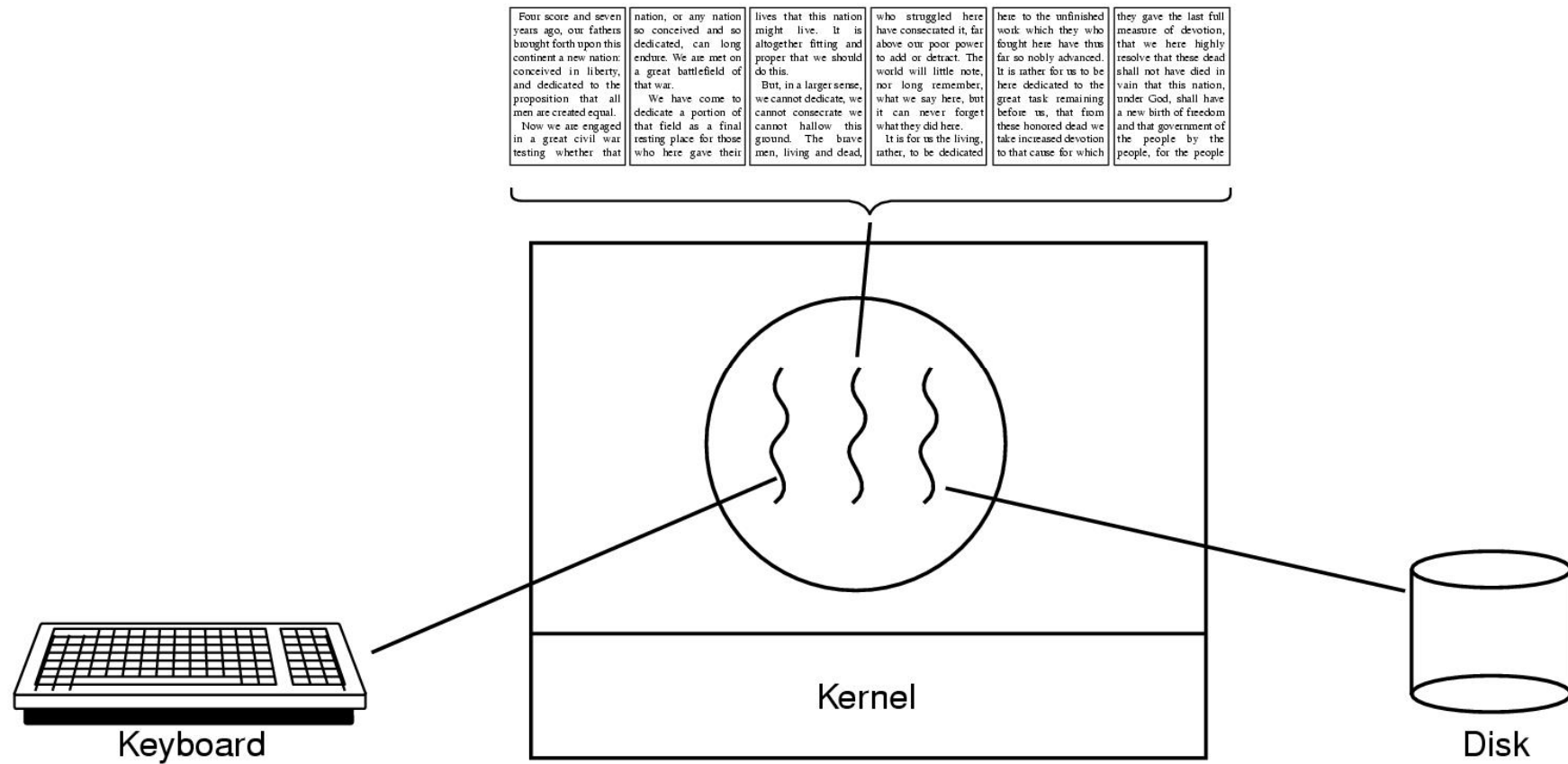
A thread of execution is the smallest sequence of programmed instructions that **can be managed independently by a scheduler**, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases **a thread is a component of a process**. Multiple threads **can exist within one process, executing concurrently and sharing resources** such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Thread

● Why do we need threads?

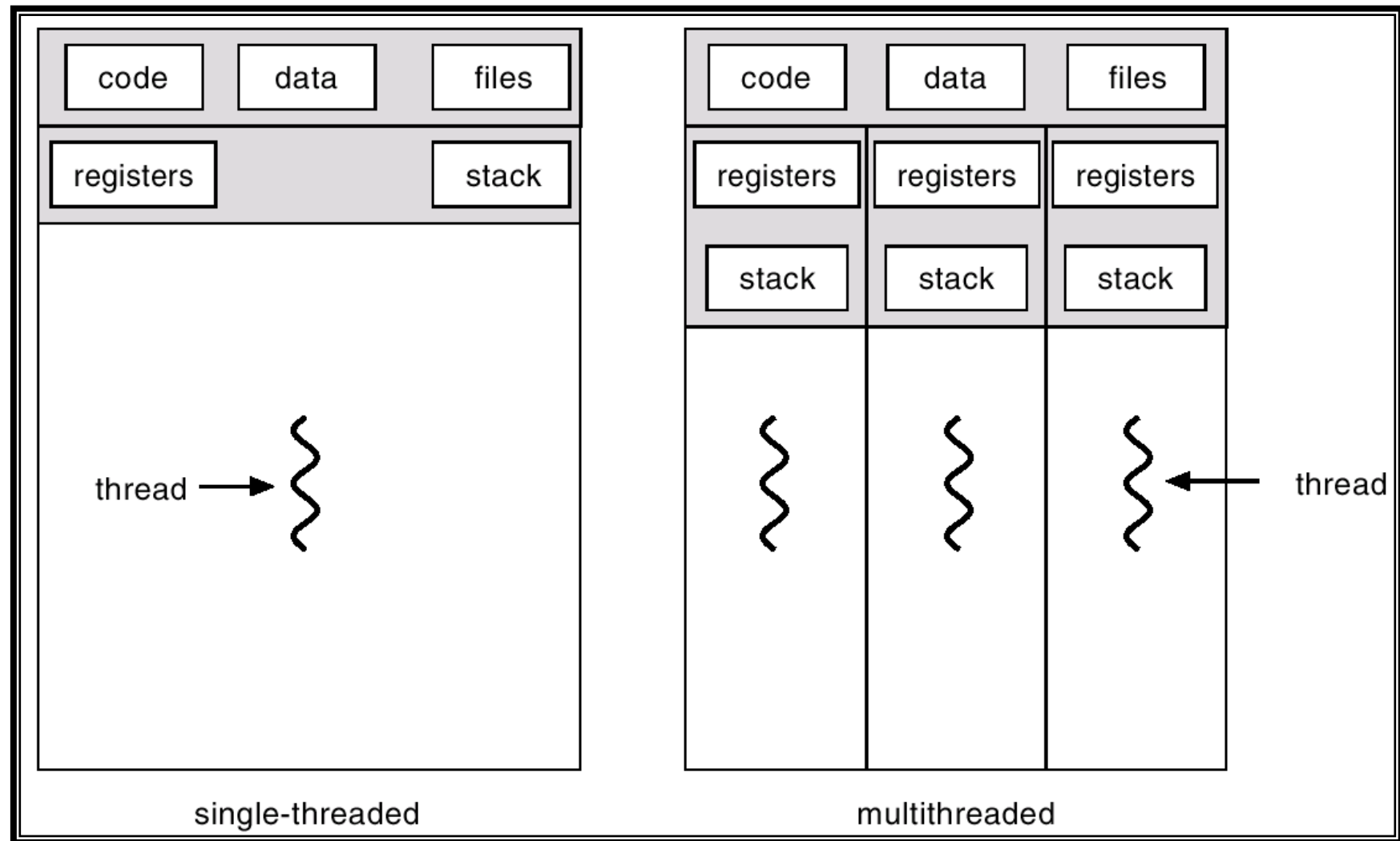
- ✓ **Responsiveness:** multiple activities can be done at the same time.
- ✓ **Resource Sharing:** threads share the memory and the resources of the process.
- ✓ **Economy:** threads are easy to create and destroy.
- ✓ **Utilization of MP (multiprocessor) Architectures:** threads are useful on multiple CPU systems.

Thread Usage Example



A word processor with three threads

Single-threaded and Multithreaded Processes



Thread Model

- Some items are shared by all threads in a process, while some items are private to each 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	

Linux: How to Create a Thread?

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
void* doSomething(void *arg)
{
    unsigned long i = 0; pthread_t id = pthread_self();
    if(pthread_equal(id,tid[0])) { printf("first\n"); }
    else { printf("Second\n");}
    return NULL;
}
int main(void) {
    int i = 0; int err;
    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL, &doSomething, NULL);
        if (err != 0) printf("error\n"); else printf("successfully\n"); i++;
    }
    return 0;
}
```



Windows: How to Create a Thread?

```
#include "windows.h"

HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // pointer to security attributes
    DWORD dwStackSize, // initial thread stack size
    LPTHREAD_START_ROUTINE lpStartAddress, // pointer to thread function
    LPVOID lpParameter, // argument for new thread
    DWORD dwCreationFlags, // creation flags
    LPDWORD lpThreadId // pointer to receive thread ID
);

e.g.,

DWORD WINAPI thread (PVOID ID) { xxx }
CreateThread(NULL, 0, thread, &ID, 0, &v);
```



Thread vs. Process

- A thread – **lightweight process**, a basic unit of CPU utilization.
- It comprises a thread ID, a program counter, a register set, and a stack.
- A traditional (**heavyweight**) process has a single thread of control.
- If the process has multiple threads of control, it can do more than one task at a time. This situation is called **multithreading**.
- Process: used to group resources together;
Thread: the entity scheduled for execution on the CPU.



POSIX Threads

- **POSIX (Portable Operating System Interface)** is a set of standard operating system interfaces based on the Unix operating system.

The need for standardization arose because enterprises using computers wanted to be able to develop programs that could be moved among different manufacturer's computer systems directly.

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

Implementation of threads

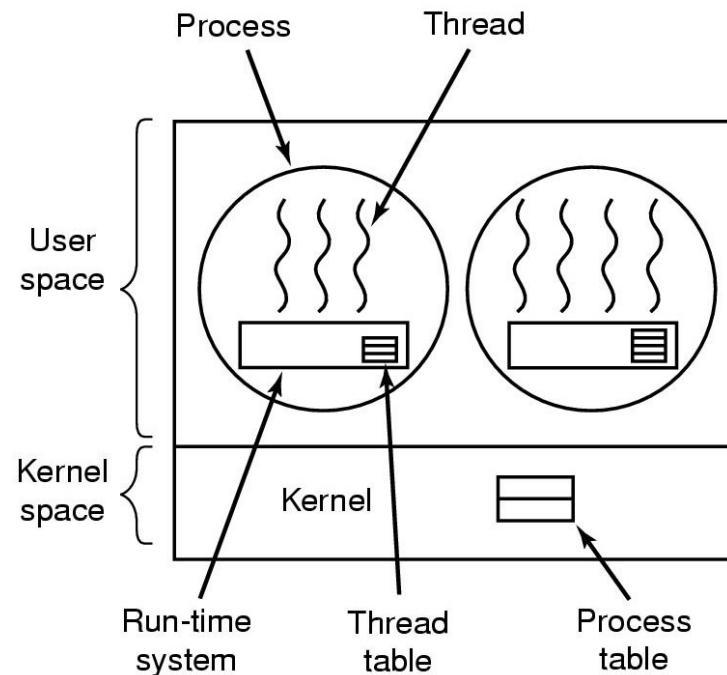
● Jigsaw reading

Briefly describe the advantages and disadvantages of the following strategies

- ① Implementing Threads in the user space (group 1 – 6)
- ② Implementing Threads in the Kernel (group 7-12)
- ③ Hybrid Implementations (group 13-18)

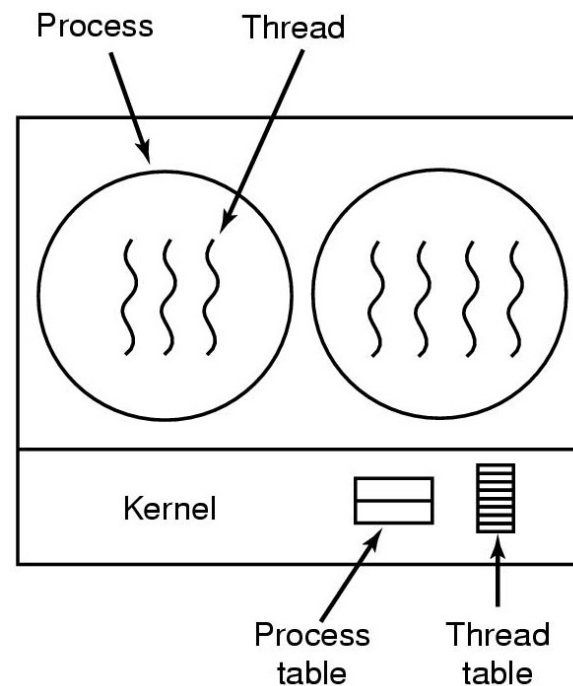
Implementing threads in user space

- Advantages: fast, flexible, scalable
- Drawbacks:
 - 1) Blocking blocks all threads in a process;
 - 2) No thread-level parallelism on multiprocessor.



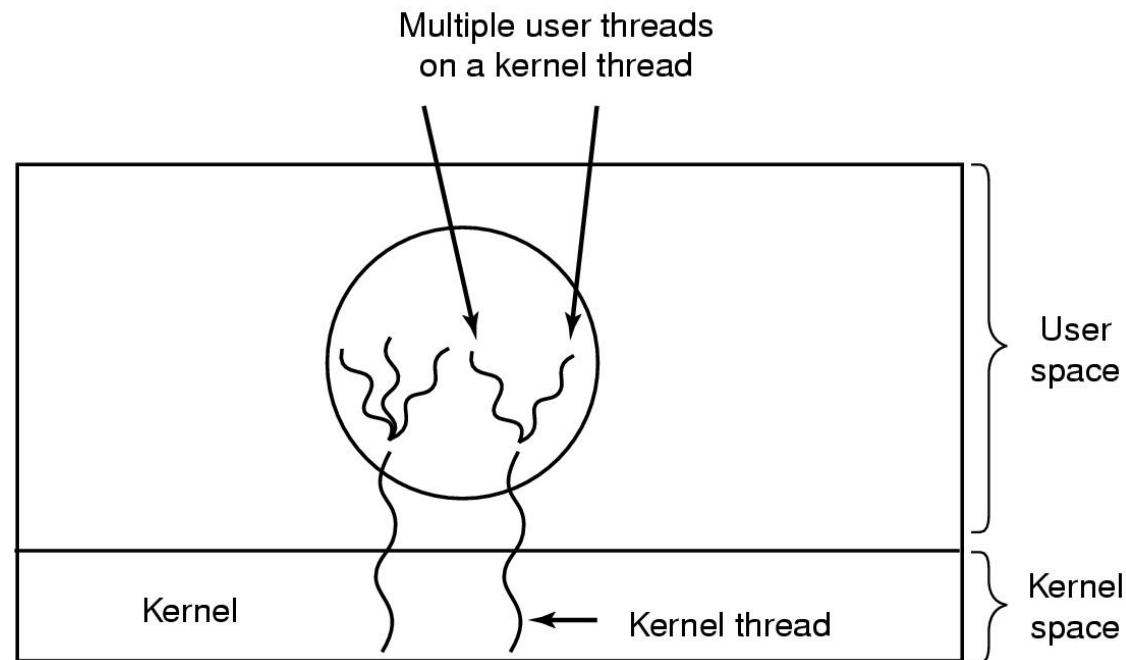
Implementing threads in kernel

- Advantages: Blocking blocks only the appropriate thread in a process; Support thread-level parallelism on multiprocessor;
- Disadvantages: Slow thread management; large thread tables



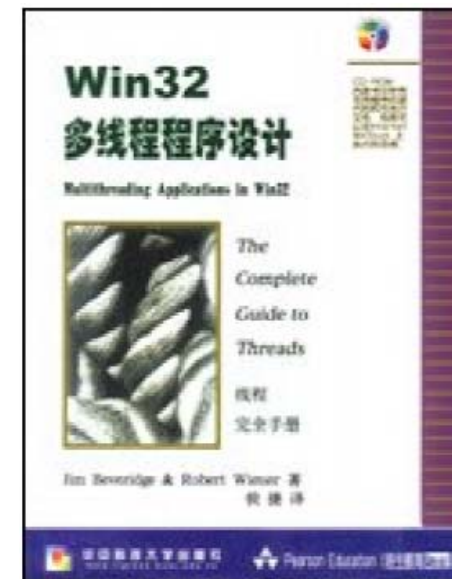
Hybrid Implementations

- Multiplex user-level threads on kernel threads. Kernel is only aware of the kernel-level threads and schedule those.
- More flexible, but more complicated.



Multithread Programming

- Multithreading applications in Win32



Check points

- How many states can a process have?
- How to create and terminate a process in Windows?
- What is PCB?
- What is Context Switch?
- What are the differences between process and thread?

