重要通知

本周起作业除非电脑坏了或者病了，作业一律不补。

- 换班或等待选课的同学请在学习通平台按时提交作业。你收到哪个班的作业通知就在哪个班交。
- 不要以我记错了时间为由晚交。请不要安排到周日再做作业，我们的作业耗时一般情况下比你估计的长很多。
- 线上 QQ 答疑时间：周五下午 2:30~4:30
- 群里可以匿名发言，同学之间可以讨论
- 答疑问题请先百度几次再来

Operating Systems

Chapter 3 Process Description and Control

A big picture:

- How to abstract program instances?
  – Process
    - Linux:
      – pstree –a　查看进程树 /windows 任务管理器
        » $ sudo apt-get install adacontrol
      – ps -aux　ppt28

- 大家一起干活 会出现啥状况
  - 场面情况： Process States
- 如何描述各自的状况
  - 各自劳动情况： Process Description
- OS 怎么管理 (创建切换) 进程
  - 管理劳动： Process Control

Process Creation

- by user:
  - New app
- By OS
  - Like process interrupt
- By existing process
  - Codeblocks runs hello.exe
  - Chrome open a new tab

Process Termination

- Processes may terminate because
  - Errors
    - Data/protection/mem failure/illegal behavior
  - Kill by parents process
  - Normal completion (job is done)

A big picture:

- How to management and share (Concurrency)
  - Information and context   (PCB)
  - communication
    - Mutual Exclusion and Synchronization/Deadlock(死锁) and Starvation
    - The SW & HW used to solve the problem (OS's perspective)
    - The classic problem and solutions (Programmer's perspective)
  - MEM CPU I/O ： MMU, Sheduling, file system

Agenda

- 3.1 What is a Process
- 3.2 Process States
- 3.3 Process Description
- 3.4 Process Control
- 3.5 Execution of the Operating System
- 3.6 process API introduction

3.1 What is a Process

Definition:

1. A program in execution
2. An instance of a program running on a computer
3. The entity that can be assigned to and executed on a processor
4. A unit of activity characterized by ：

   ① the execution of a sequence of instructions

   ② a current state

   ③ an associated set of system resources

3.1 What is a Process

- Process Elements
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers
  - Context data
  - I/O status information
  - Accounting information

Agenda

- 3.1 What is a Process
- <u>3.2 Process States</u>
- 3.3 Process Description
- 3.4 Process Control
- 3.5 Execution of the Operating System
- 3.6 process API introduction

3.2 Process States

- 3.2.1 Trace of the Process
- 3.2.2 A Two-State Process Model
- 3.2.3 A Five-State Model
- 3.2.4 Suspended Process

3.2 Process States

- How to Inspect Multiple processes
  - By users:
    - Trace of Process(进程轨迹)
  - By OS:
    - Linked lists according to different States of Processes

3.2 Process States

Trace of Process(进程轨迹)

- Sequence of instruction that execute for a process
- Dispatcher(调度器) switches the processor from one process to another

3.2 Process States

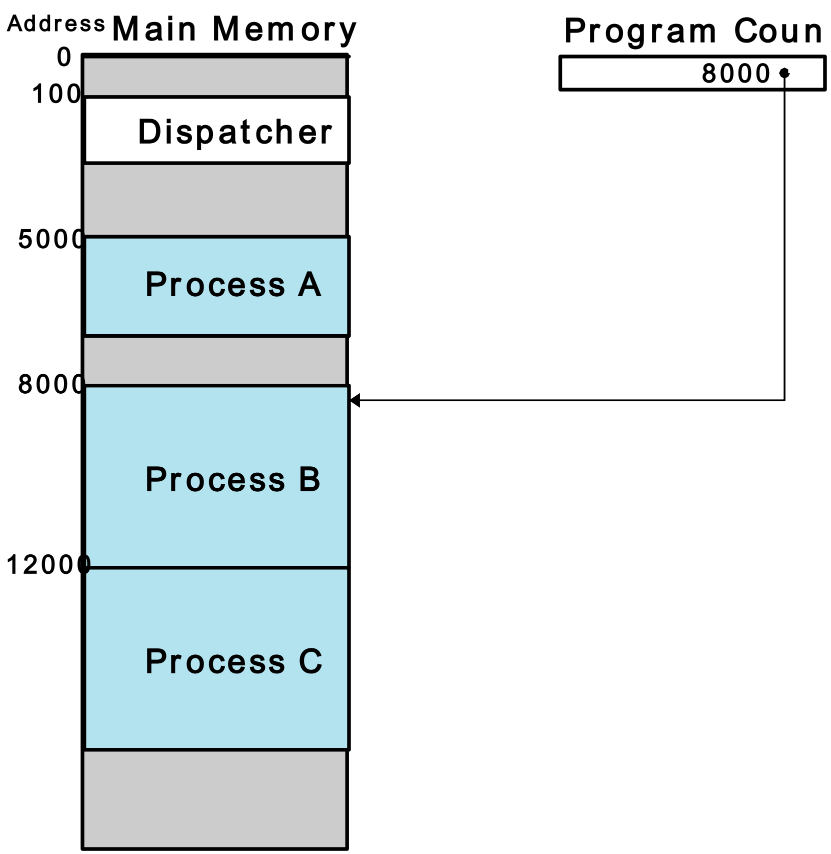


Figure 3.2 Snapshot of Example Execution at Instruction Cycle 13

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------------- Time out	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
-------------- I/O request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003

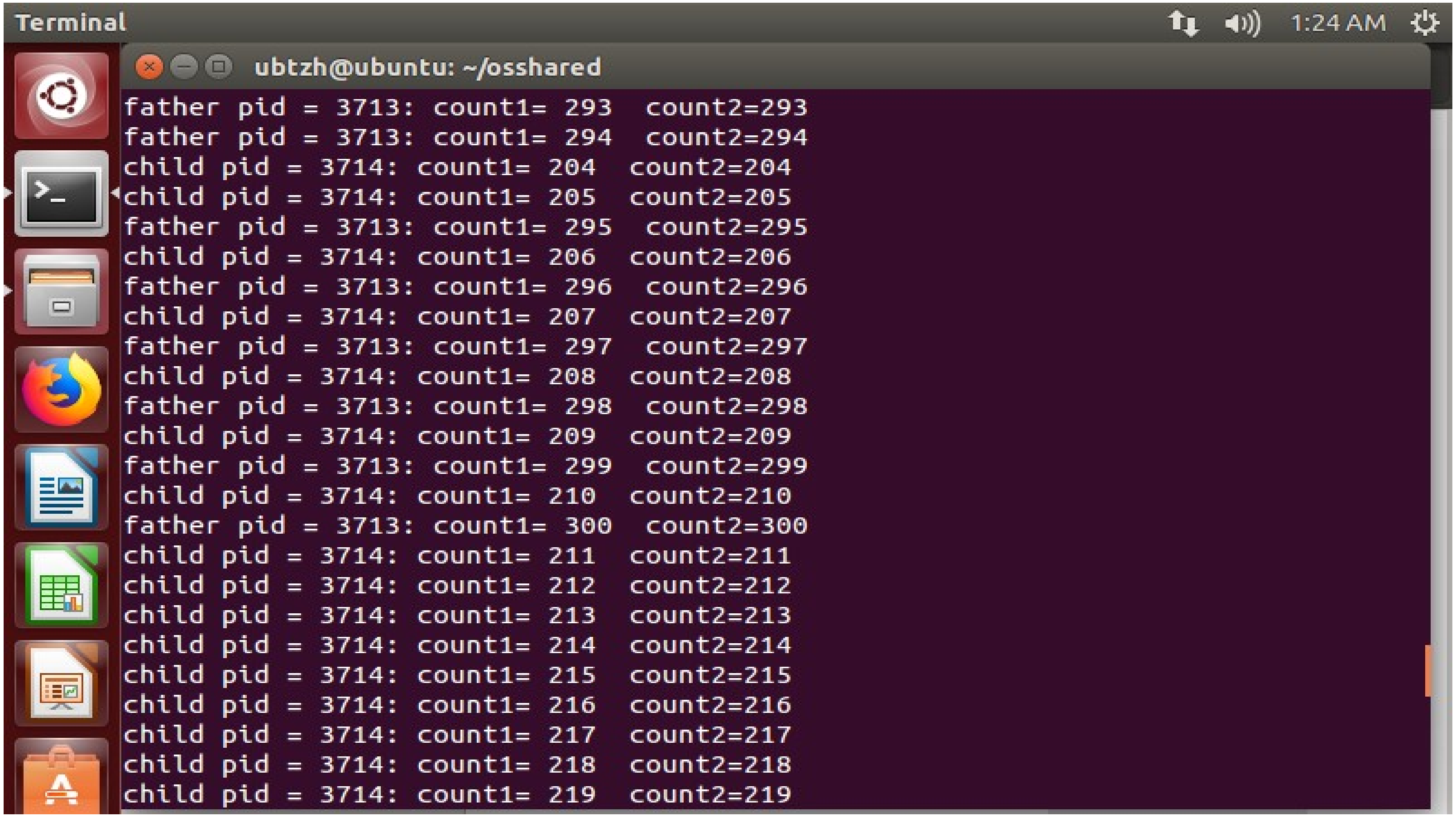
27	12004
28	12005
----------------- Time out	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------------- Time out	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------------- Time out	

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

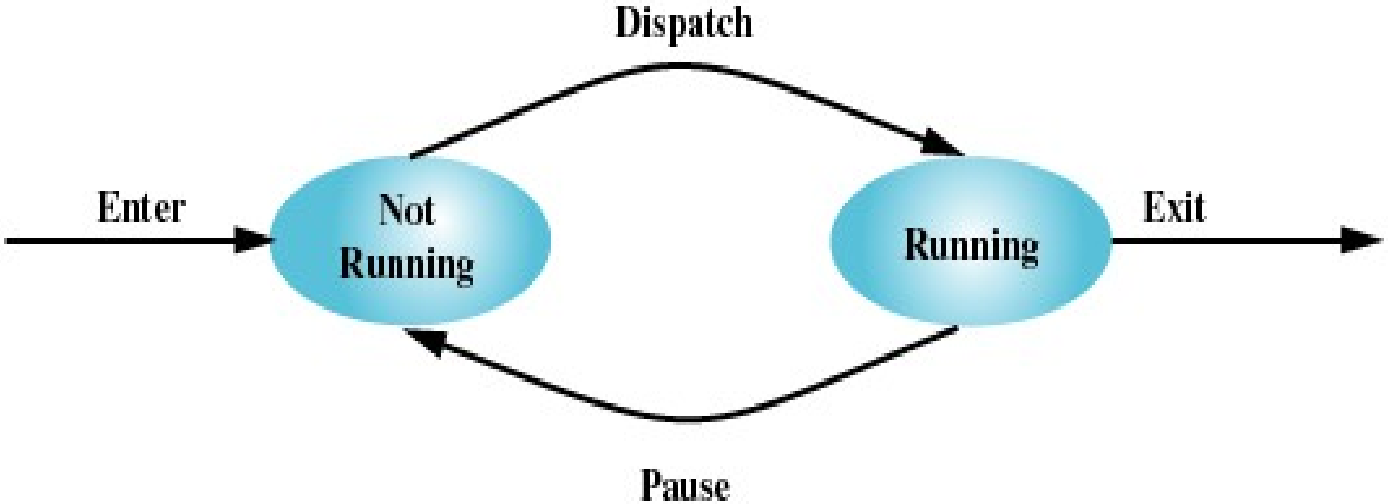
3.2 Process States



3.2 Process States

Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running

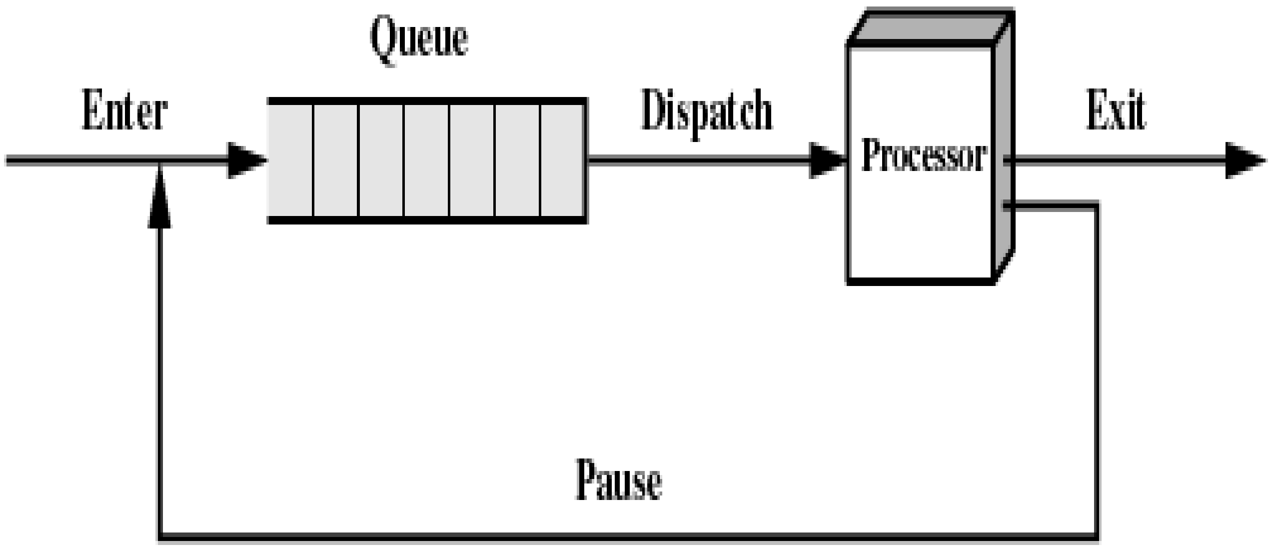


(a) State transition diagram

3.2 Process States

Two-State Process Model

Two states , one queue, any disadvantage ?



(b) Queuing diagram

3.2 Process States

- Limit of Two-State Process Model
  - Dispatcher 分派器 cannot just select the process that has been in the queue the longest because it may be blocked
  - Not-running
    - ready to execute 就绪
    - waiting for I/O (blocked 阻塞)

3.2 Process States

A Five-State Model

- Running(运行态)
- Ready （ 就绪态)
- Blocked(阻塞态)
- New （ 新建态)
- Exit （ 退出态)

A Five-State Model

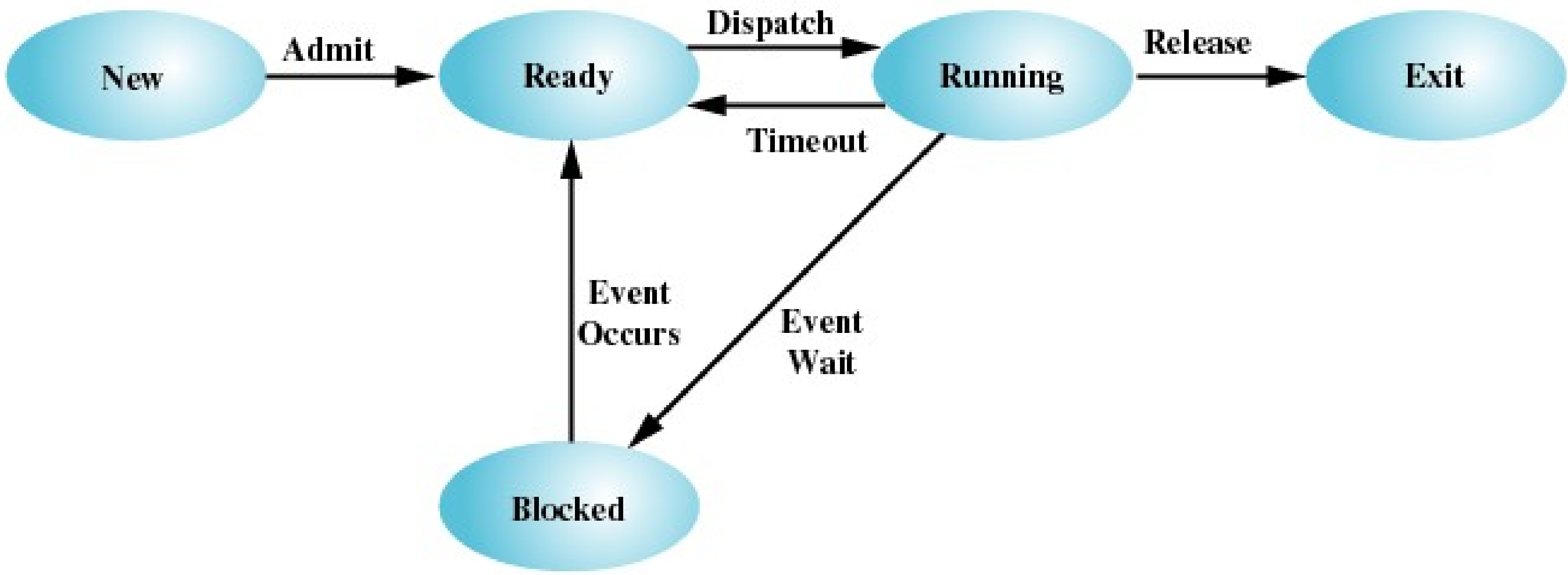


Figure 3.6 Five-State Process Model

3.2 Process States

A Five-State Model

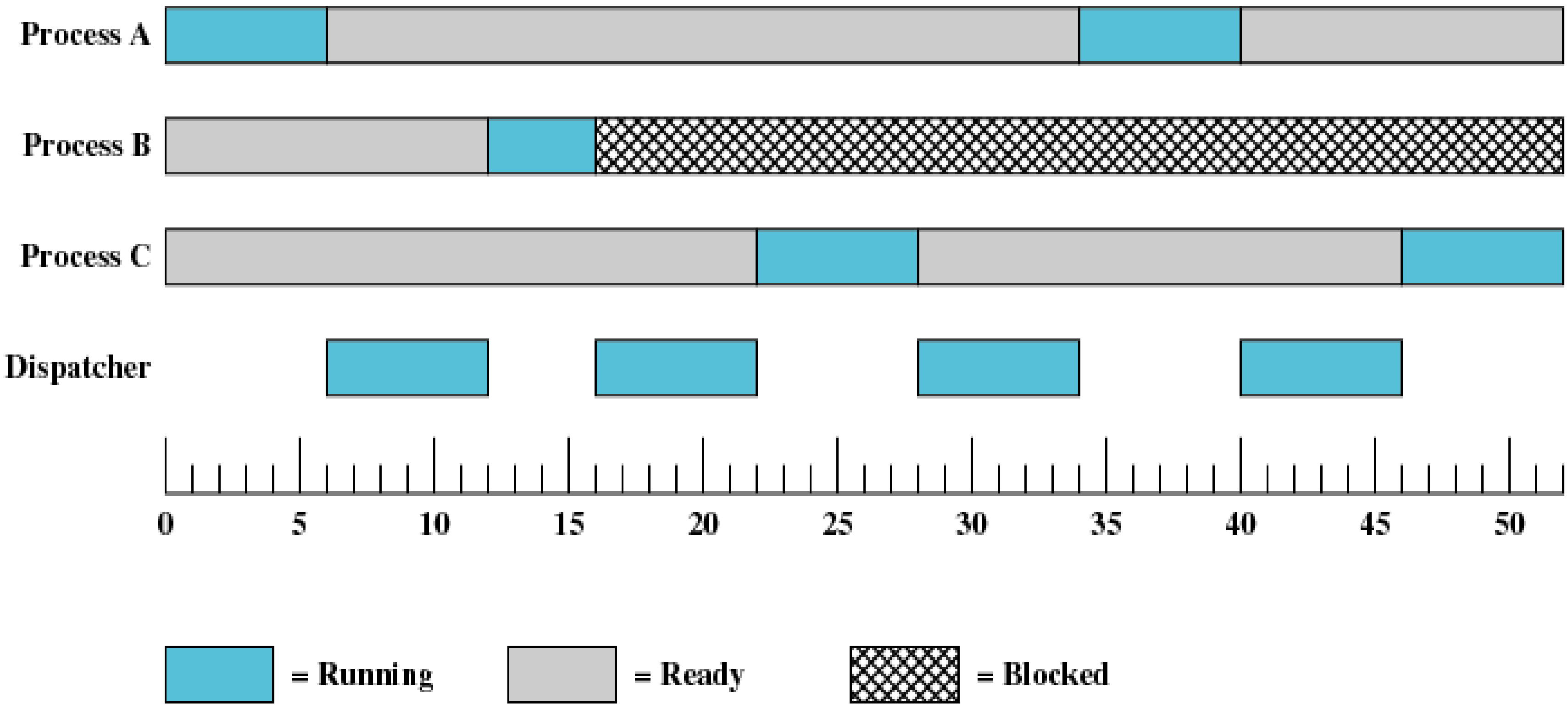
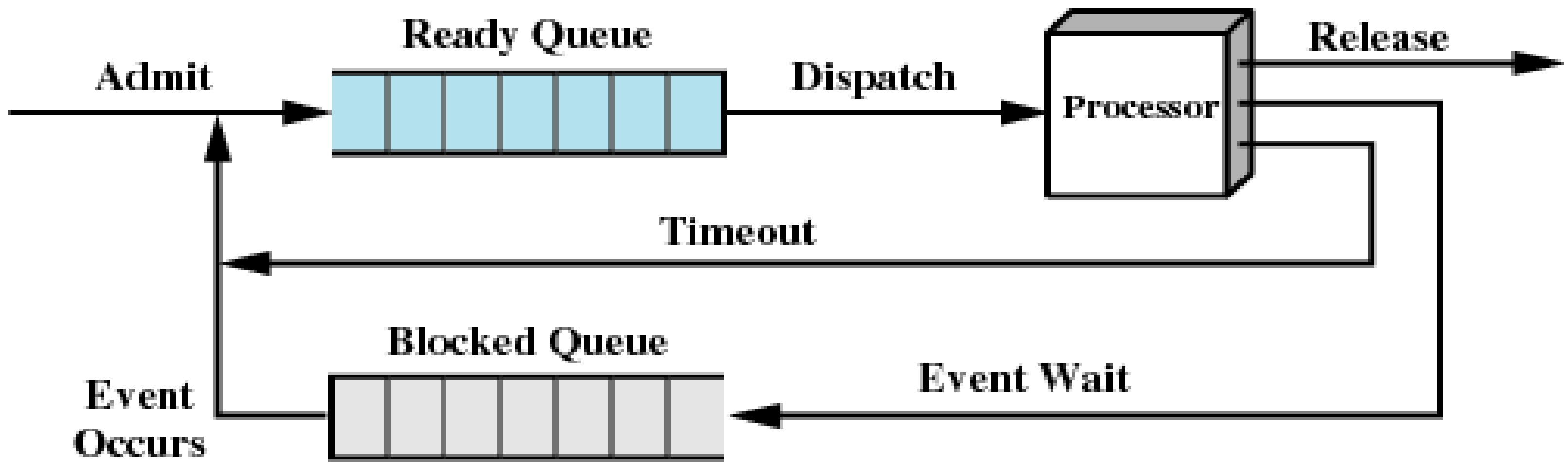


Figure 3.7 Process States for Trace of Figure 3.4

3.2 Process States

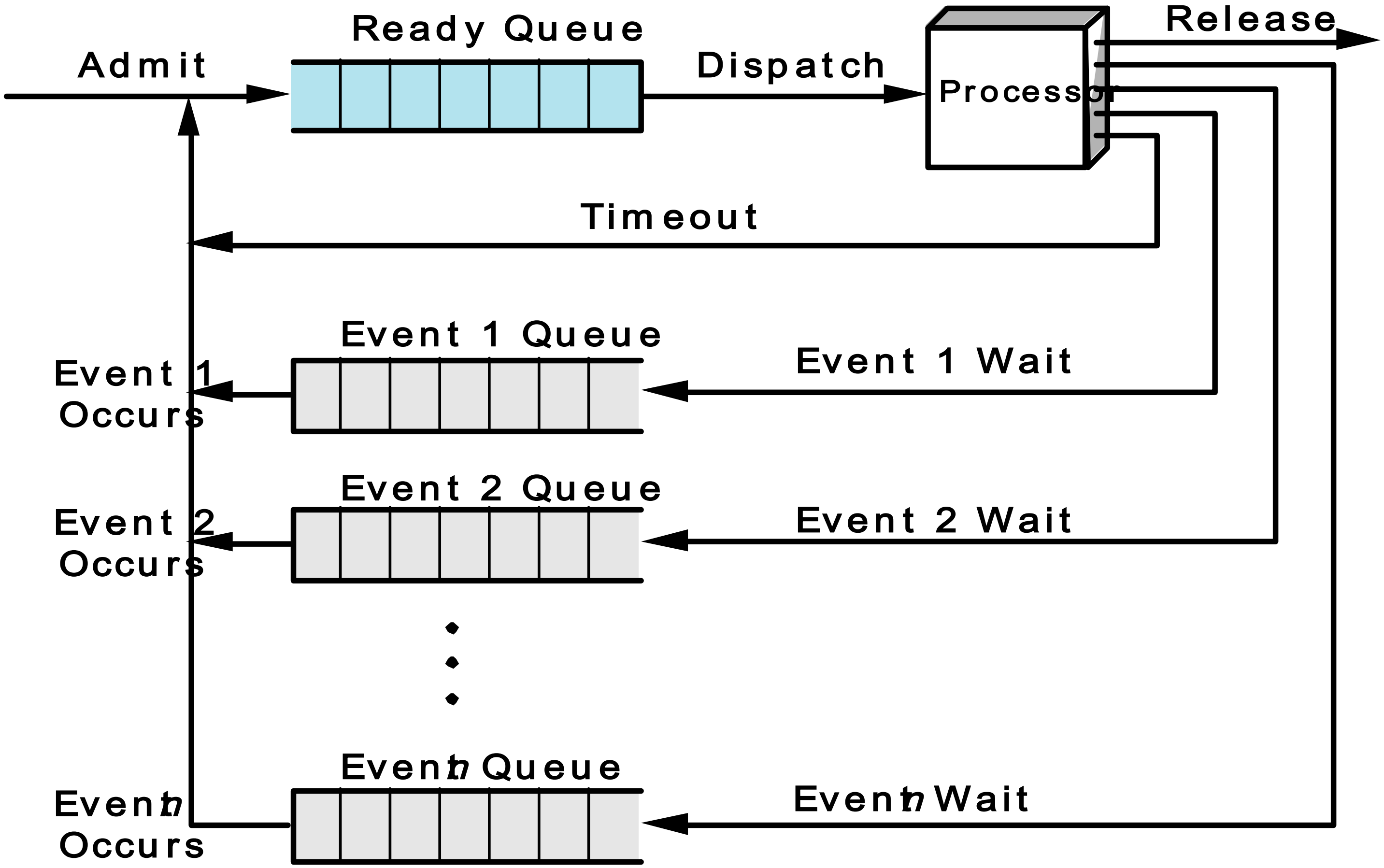
A Five-State Model

Using Two Queues ： Efficiency ？



(a) Single blocked queue

Multiple Blocked Queues



(b) Multiple blocked queues

A Five-State Model

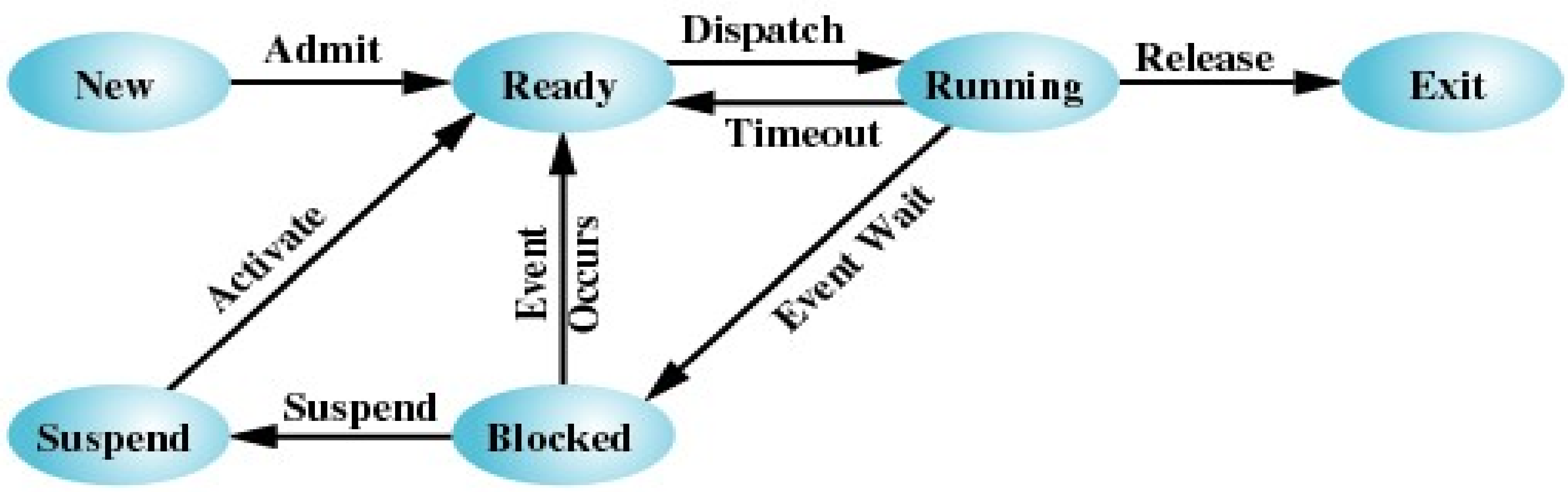
- Is it perfect?
  - So far, we only focus on all the possible states of processes, together with the resources they may waiting for.
  - Now consider the **memory spaces** allocated for all the processes
    - Any upper limitation of the number of process?
    - Less impact on capability of the computer system

3.2 Process States

Suspended Processes(被挂起的进程)

- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
- Two new states
  - Blocked/Suspend
  - Ready/Suspend

A Five-State Model+One Suspend State



(a) With One Suspend State

A Seven-State Model(Two Suspend States)

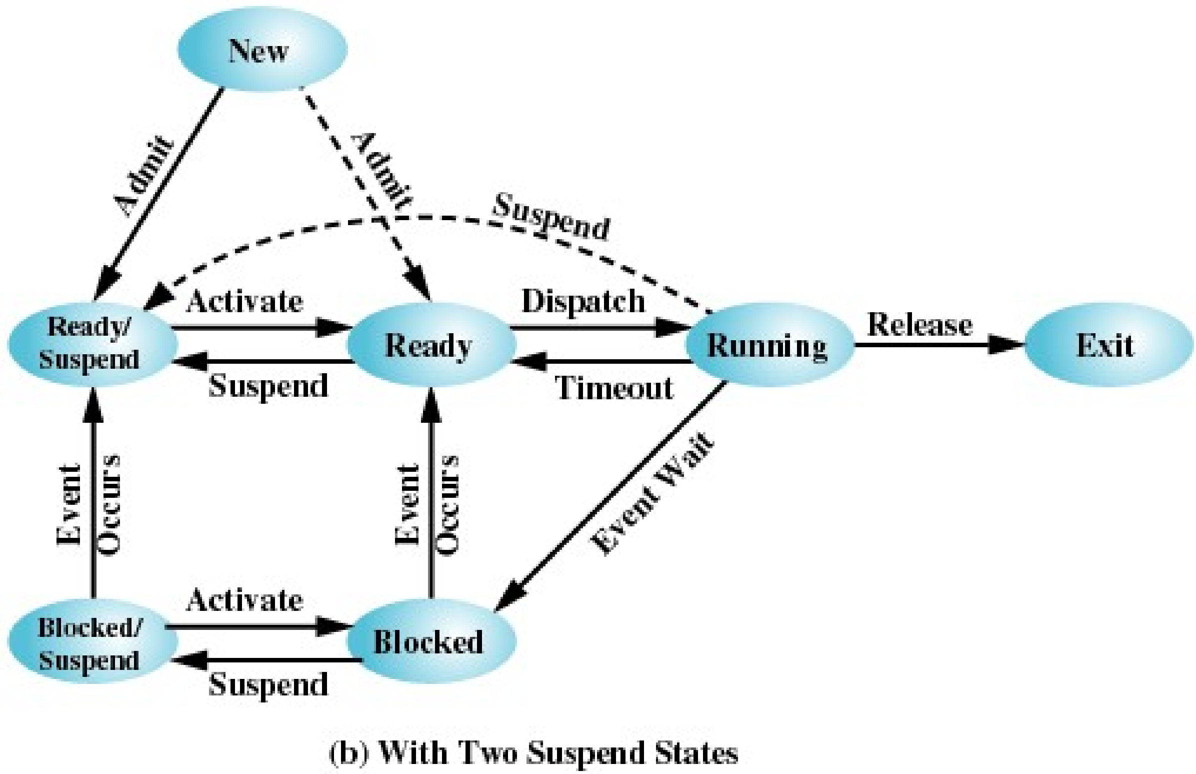


Figure 3.9 Process State Transition Diagram with Suspend States

Linux 进程状态

- R （TASK_RUNNING），可执行状态（对应就绪和执行）
- S （TASK_INTERRUPTIBLE），可中断的睡眠状态，如等待信号量
- D （TASK_UNINTERRUPTIBLE），不可中断的睡眠状态，不受异步信号唤醒（少见）进程必须等待直到有中断发生
- T （TASK_STOPPED or TASK_TRACED），暂停状态或跟踪状态，gdb 断点调试
- Z （TASK_DEAD - EXIT_ZOMBIE），退出状态，进程成为僵尸进程，仅保留 task_struct(PCB），其它的都释放了，等待父进程 wait 来释放

<	高优先级
N	低优先级
L	有些页被锁进内存
s	包含子进程
+	位于后台的进程组；
l	多线程，克隆线程

Linux 进程状态

- ps –aux   ps -ef

```
ubtzh@ubuntu: ~
ubtzh@ubuntu:~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.2  33924  2612 ?        Ss   Mar11   0:07 /sbin/init
root           2  0.0  0.0      0     0 ?        S    Mar11   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    Mar11   0:02 [ksoftirqd/0]
root           5  0.0  0.0      0     0 ?        S<   Mar11   0:00 [kworker/0:0H]
root           7  0.0  0.0      0     0 ?        S    Mar11   0:15 [rcu_sched]
root           8  0.0  0.0      0     0 ?        S    Mar11   0:00 [rcu_bh]
root           9  0.0  0.0      0     0 ?        S    Mar11   0:00 [migration/0]
root          10  0.0  0.0      0     0 ?        S    Mar11   0:00 [watchdog/0]
root          11  0.0  0.0      0     0 ?        S    Mar11   0:00 [kdevtmpfs]
root          12  0.0  0.0      0     0 ?        S<   Mar11   0:00 [netns]
root          13  0.0  0.0      0     0 ?        S<   Mar11   0:00 [perf]
root          14  0.0  0.0      0     0 ?        S    Mar11   0:00 [khungtaskd]
root          15  0.0  0.0      0     0 ?        S<   Mar11   0:00 [writeback]
root          16  0.0  0.0      0     0 ?        SN   Mar11   0:00 [ksmd]
```

Agenda

- 3.1 What is a Process
- 3.2 Process States
- <u>3.3 Process Description</u>
- 3.4 Process Control
- 3.5 Execution of the Operating System
- 3.6 process API introduction

3.3 Process Description

- To answer a fundamental question: What information does the OS need to control processes and manage resources for them?

  - 3.3.1 <u>Operating System Control Structures</u>
    - 即： OS 掌握进程哪些信息，怎么存储这些信息
  - 3.3.2 Process Control Structures
    - 即：进程记录哪些信息便于运行和管理

3.3.1 Operating System Control Structures

- Information about the current status of each process and resource（每个进程和资源的当前状态）
- Tables are constructed for manage 4 kinds of resources（操作系统构造并维护他所管理的四类资源实体的信息表）
  - MEM,I/O,FILE,PROCESS
  - Tables are linked or cross-referenced 这些表交叉引用

- 3.3.1 Operating System Control Structures

OS 控制表通用结构

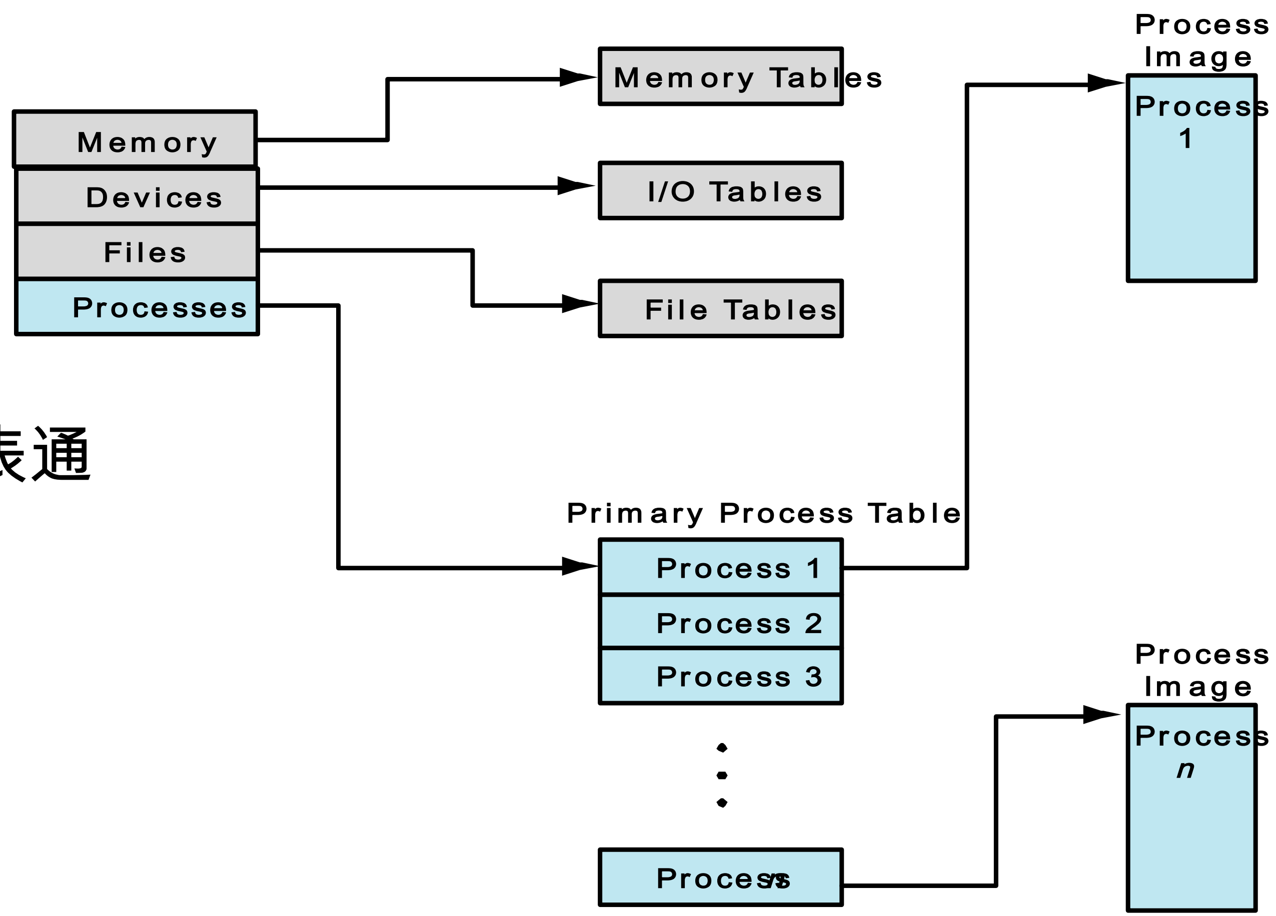


Figure 3.11 General Structure of Operating System

3.3.1 Operating System Control Structures

- Memory Tables
  - Allocation of main memory to processes (分配给进程的主存)
  - Allocation of secondary memory to processes (分配给进程的辅存)
  - Protection attributes for access to shared memory regions(共享内存区域的保护属性)
  - Information needed to manage virtual memory(虚拟内存的管理信息)

3.3.1 Operating System Control Structures

- I/O Tables
  - I/O device is available or assigned(分配状态)
  - Status of I/O operation
  - Location in main memory being used as the source or destination of the I/O transfer (数据传送的源和目的地址)

3.3.1 Operating System Control Structures

- File Tables
  - Existence of files
  - Location on secondary memory
  - Current Status
  - Attributes

3.3 Process Description

- 3.3.1 Operating System Control Structures
- 3.3.2 Process Control Structures

3.3.2 **Process Control Structures**

- Process image 进程映像
  - The collection of program, data, stack, and attributes(PCB): not contiguous in addresses

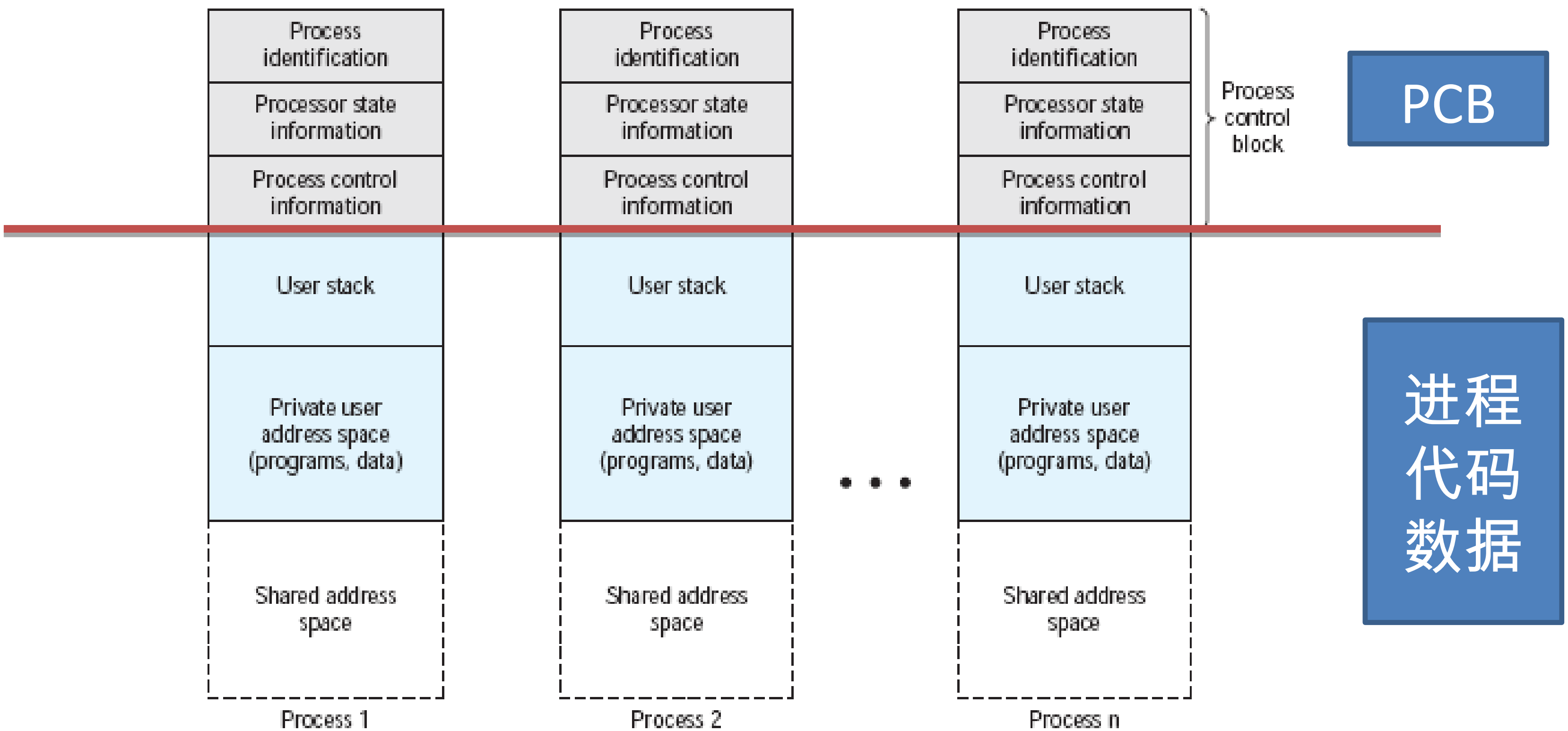


Figure 3.13 User Processes in Virtual Memory

3.3.2 **Process Control Structures**

in memory tables by OS **VS** **in PCB by Process**

- Ppt34 页内容
- show the location of each page of each process image. (页表)
- Process Attributes
- the location
  - on disk
  - and in main memory

- Textbook Table 3.5
  - process identifier
  - Processor state information
    - 切换进程需要
  - Process control information

3.3.2 **Process Control Structures**

- Process Control Block
  - Process Identification
  - Processor State Information
    - User-Visible Registers
    - Control and Status Registers
    - Stack Pointers
  - Process Control information
    - Scheduling and State Information
    - Data Structuring (link information)
    - Interprocess Communication
    - Process Privileges
    - Memory Management
    - Resource Ownership and Utilization

3.3.2 Process Control Structures

PCB contains: Structuring information: pointers to linked list of queues

Any potential risk?

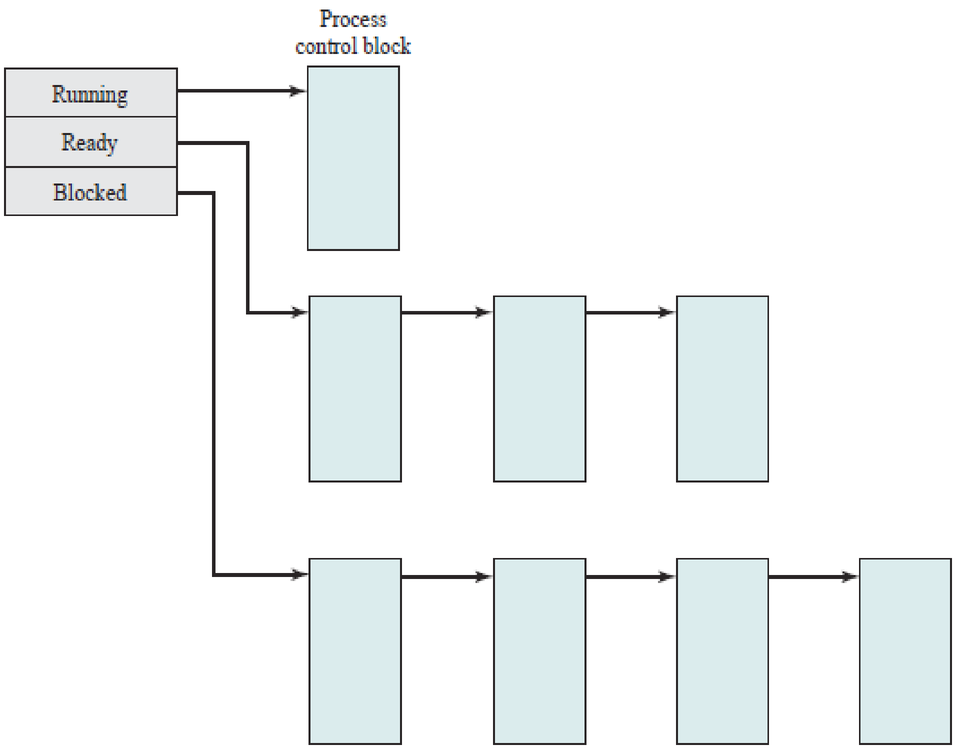


Figure 3.14 Process List Structures

Agenda

- 3.1 What is a Process
- 3.2 Process States
- 3.3 Process Description
- <u>3.4 Process Control</u>
- 3.5 Execution of the Operating System
- 3.6 process API introduction

3.4 Process Control

- 3.4.1 Modes of Execution
- 3.4.2 Process Creation
- 3.4.3 Process Switching

3.4.1 Modes of Execution(CPU)

- To protect OS
  - User mode
    - Typically when User programs executes
  - System mode, control mode, or kernel mode
    - More-privileged mode
    - Kernel of the operating system
    - Table 3.7

3.4.1 Modes of Execution(CPU)

- Translation between the two model
  - Interrupt/Trap/Syscall
  - Modifiy Processor status register (psr) and current privileged level (cpl)

Figure 3.4

3.4.1 Modes of Execution(CPU)

Table 3.7
Typical Functions
of an OS Kernel

System mode

3.4.2 Process Creation

Process Creation

1. Assign a unique process identifier
2. Allocate space for the process
3. Initialize process control block
4. Set up appropriate linkages
   - Ex: add new process to linked list used for scheduling queue
5. Create of expand other data structures
   - Ex: maintain an accounting file

3.4.3 Process Switching

Process Switching ： When

A process switch may occur any time that the OS has gained control from the currently running process. The possible events that may give control to the OS include:

1. Interrupt
   - Clock interrupt
     - ✓ process has executed for the maximum allowable time slice
   - I/O interrupt
   - Memory fault
     - ✓ Referenced virtual address is not in main memory, so it must be brought in.

3.4.3 Process Switching

2. Trap
   - error or exception occurred
   - may cause process to be moved to Exit state
3. Supervisor call (System Call)
   - such as file open

3.4.3 Process Switching

Process Switching ： How

1. Save context of processor including program counter and other registers
2. Update the process control block of the process and change the process's state that is currently in the Running state
3. Move process control block to appropriate queue – ready; blocked; ready/suspend
4. Select another process for execution

3.4.3 Process Switching

Process Switching ： How

5. Update the process control block of the process selected and change it state
6. Update memory-management data structures
7. Restore context of the selected process

Agenda

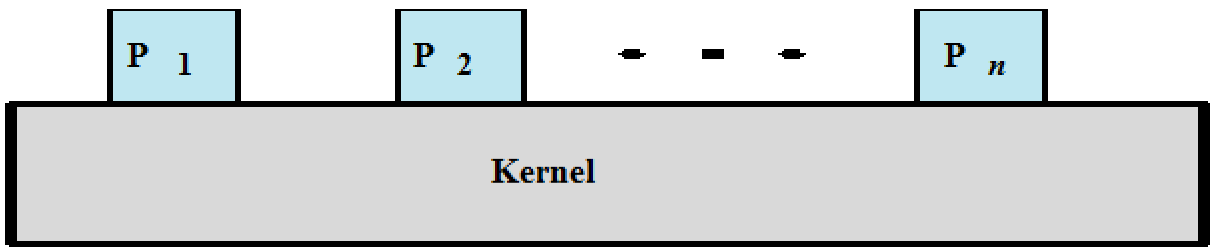
- 3.1 What is a Process
- 3.2 Process States
- 3.3 Process Description
- 3.4 Process Control
- <u>3.5 Execution of the Operating System</u>
- 3.6 process API introduction

3.5 Execution of the Operating System

- 1.Non-process Kernel （无进程内核）
- 2.Execution Within User Processes(在用户进程中执行)
- 3.Process-Based Operating System （基于进程的OS ）

3.5 Execution of the Operating System

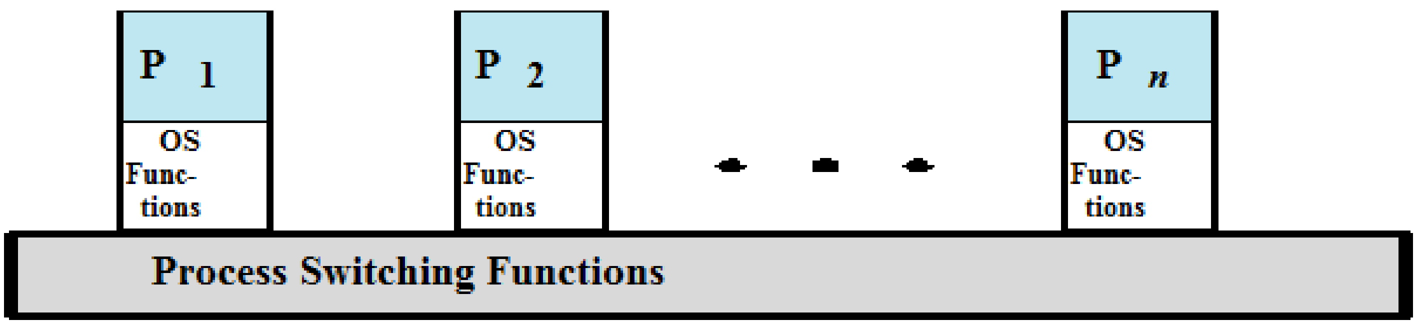
1. Non-process Kernel（无进程内核）
   - Execute kernel outside of any process
   - Operating system code is executed as a separate entity that operates in privileged mode



(a) Separate kernel

3.5 Execution of the Operating System

2. Execution Within User Processes(在用户进程中执行)
   - Operating system software within context of a user process
   - Process executes in privileged mode when executing operating system code
   - Unix



(b) OS functions execute within user processes

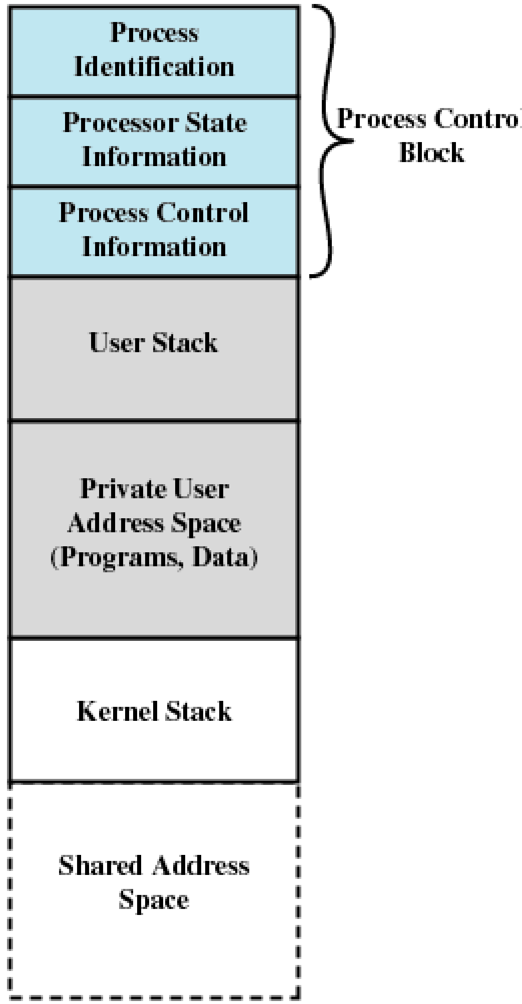
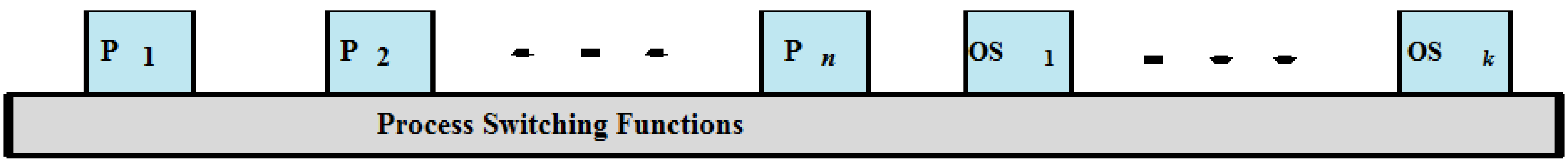


Figure 3.16 Process Image: Operating System Executes Within User Space

3.5 Execution of the Operating System

3. Process-Based Operating System（基于进程的 OS ）
   - Implement operating system as a collection of system processes
   - Useful in multi-processor or multi-computer environment



(c) OS functions execute as separate processes

3.6 process API introduction

- 3.1 What is a Process
- 3.2 Process States
- 3.3 Process Description
- 3.4 Process Control
- 3.5 Execution of the Operating System
- <u>3.6 process API introduction</u>

3.6 process API introduction(1/15)

- 创建 create
  - fork
    - 在父进程中，fork 返回新创建子进程的进程 ID ；
    - 在子进程中，fork 返回 0 ；
    - 如果出现错误，fork 返回一个负值；
- 销毁 destroy
- 等待 wait

3.6 process API introduction(2/15)

- fork

```
//main.c
int main()
{
   printf("first command:(pid:%d)\n",(int)getpid());

   int   rc=fork();

   if(rc)
      printf("%d father of %d\n",(int)getpid(),rc);
   else
      printf("%d child\n",(int)getpid());
   return 0;
}
```

Process image

PCB ： parent
User stack: rc 非零，为子进程 pid
Code & data: main
…

Process image

PCB ： child
User stack ： rc 为 0
Code & data 代码内容同上
…

3.6 process API introduction(3/15)

- fork

parent

```
int main()
{
   printf("first command:(pid:%d)\n",(int)getpid());

   int   rc=fork();

   if(rc)
      printf("%d father of %d\n",(int)getpid(),rc);
   else
      printf("%d child\n",(int)getpid());
   return 0;
}
```

Why clone the same code?

3.6 process API introduction(4/15)

- fork

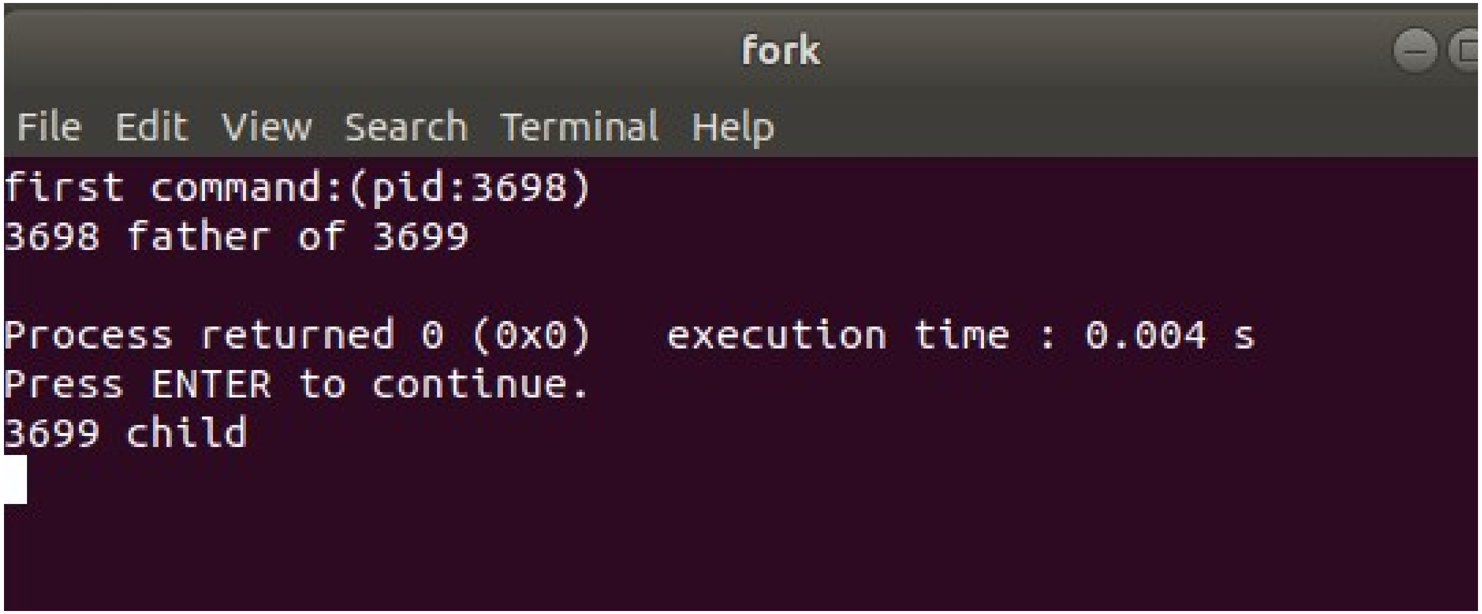
child

```
int main()
{
    ~~printf("first command:(pid:%d)\n",(int)getpid());~~// 机器指令存在，但不执行

    int   rc=~~fork();~~ // 返回值保存执行，但 fork 的功能不执行

    if(rc)
        printf("%d father of %d\n",(int)getpid(),rc);
    else
        printf("%d child\n",(int)getpid());
    return 0;
}
```

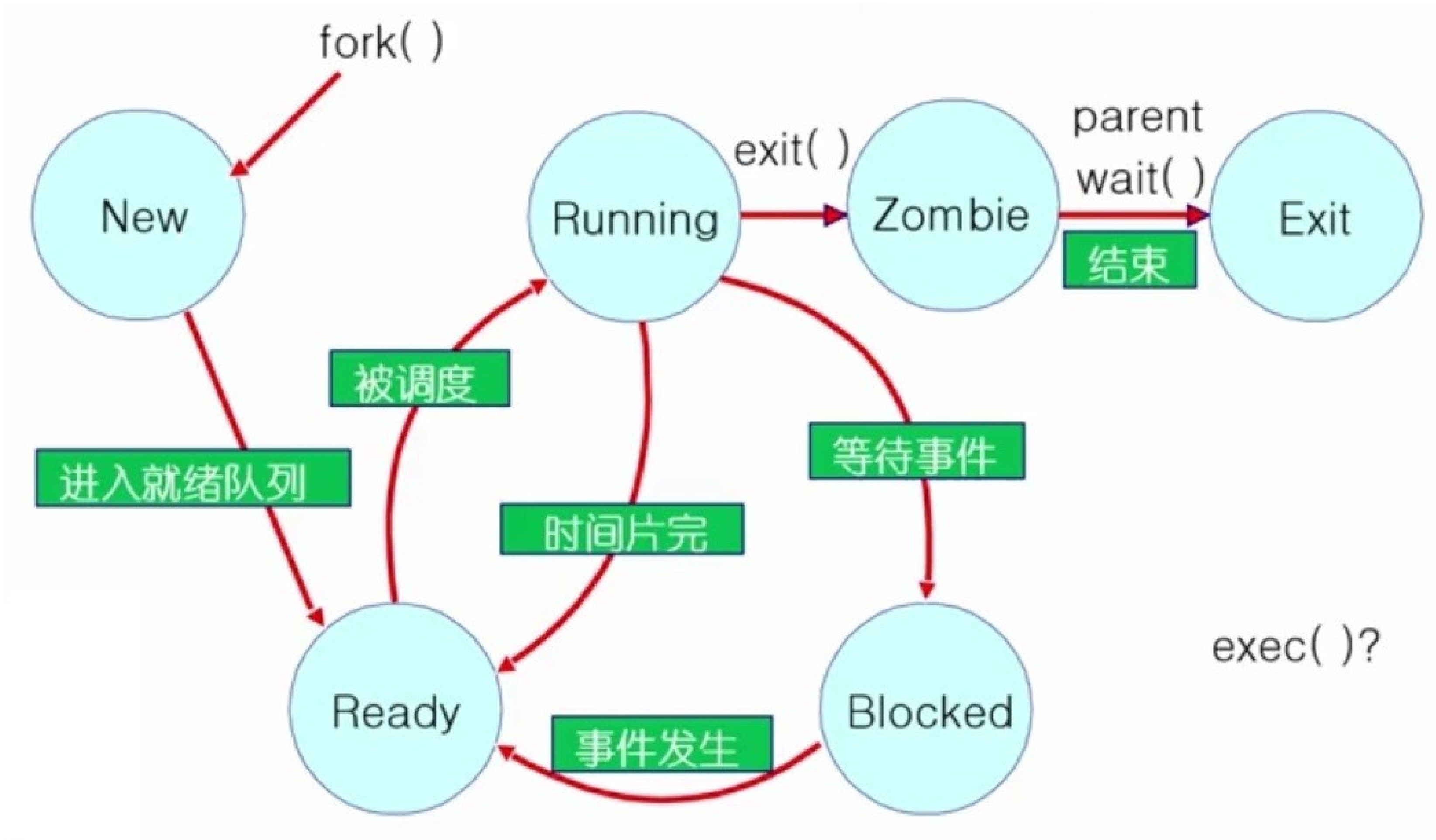
3.6 process API introduction(5/15)



3.6 process API introduction(6/15)

- wait: 父进程等待子进程终止并返回
- why ：进程只能释放自己的空间，不能销毁自己的 PCB ，父进程销毁子进程的 PCB
- 孤儿进程：父进程已经结束，子进程尚未结束

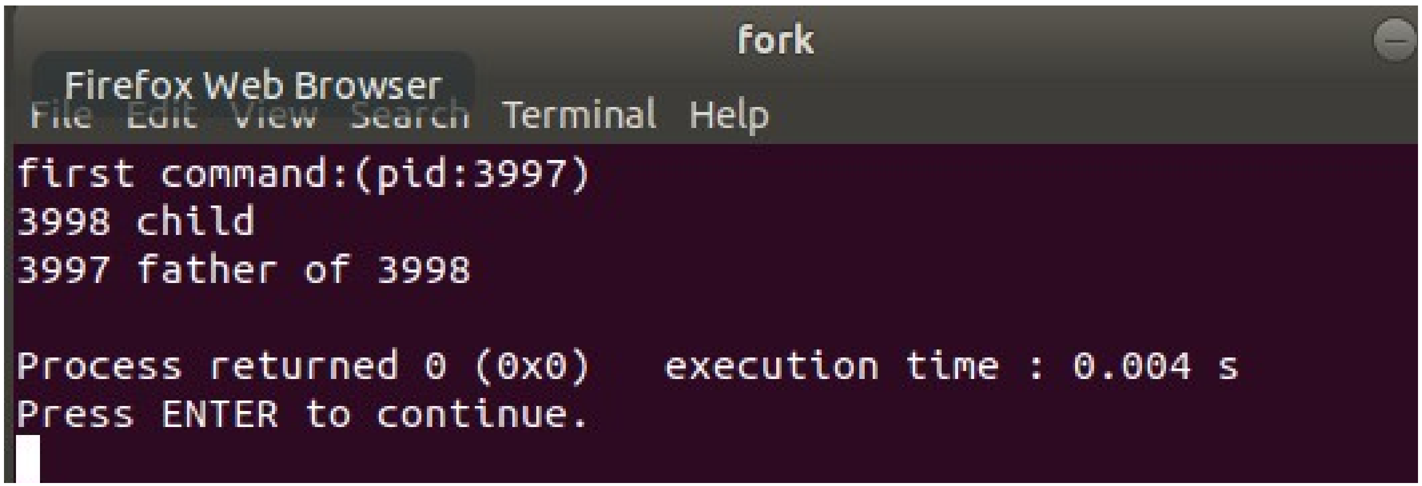
3.6 process API introduction(7/15)



3.6 process API introduction(8/15)

- int main()
- {
- int wc ;
- printf("first command:(pid:%d)\n",(int)getpid());
- int rc=fork();
- if(rc) {
- wait(& wc);
- printf("%d father of %d\n",(int)getpid(),rc);
- }
- else
- printf("%d child\n",(int)getpid());
- return 0;
- }

3.6 process API introduction(9/15)



3.6 process API introduction(10/15)

- exec(lab04:3.4.3)

```
//execDemo.c
int main()
{
    fpid=fork() ;
char *args[]={"./EXEC",NULL};
  if(fpid==0)
    execvp(args[0],args);
  printf("Ending-----");
  return 0;
}
```

执行完 fork() 时，各个进程映像

PCB　: parent	
User stack:	args
Code & data　:	**execDemo**
…	

Process　image

PCB　: child	
User stack:	args
Code & data　:	**execDemo**
…	

3.6 process API introduction(11/15)

```c
//execDemo.c
int main()
{
    fpid=fork();
char *args[]={"./EXEC",NULL};
   if(fpid==0)
      execvp(args[0],args);
   printf("Ending-----");
   return 0;
}
```

执行完 execvp() 时，各个进程映像

PCB　：parent
User stack: args
Code & data : exeDemo
…

PCB ： child
User stack ： i
Code & data ： EXEC
…

```c
//EXEC.c
int main()  {
   int i;
   printf("I am EXEC.c called by execvp() ");
return 0;
}
```

3.6 process API introduction(12/15)

```
int main()
{
    printf("first command:(pid:%d)\n",(int)getpid());
    int   rc=fork();
    char *args[]={"./exec",NULL};
    if(rc) {
        printf("%d father of %d\n",(int)getpid(),rc);
         int wc=wait();
    }
    else{
        execvp(args[0],args);
            printf("%d child\n",(int)getpid());// 不会执行
    }
    return 0;
}
```

3.6 process API introduction(13/15)

```c
//EXEC.c
int main()  {
    int i;
    printf("I am EXEC.c called by execvp() ");
return 0;
}
```

```
first command:(pid:4866)
4866 father of 4867
I am EXEC.c called by execvp()
Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
```

3.6 process API introduction(14/15)

```
//execDemo.c
int main()
{
char *args[]={"./EXEC",NULL};
execvp(args[0],args);
   printf("Ending-----");
   return 0;
}
```

PCB :
User stack : i
Code & data : EXEC
…

```
//EXEC.c
int main()  {
   int i;
   printf("I am EXEC.c called by execvp() ");
return 0;
}
```

3.6 process API introduction(15/15)