Process Concept — Chapter 3

2022年9月

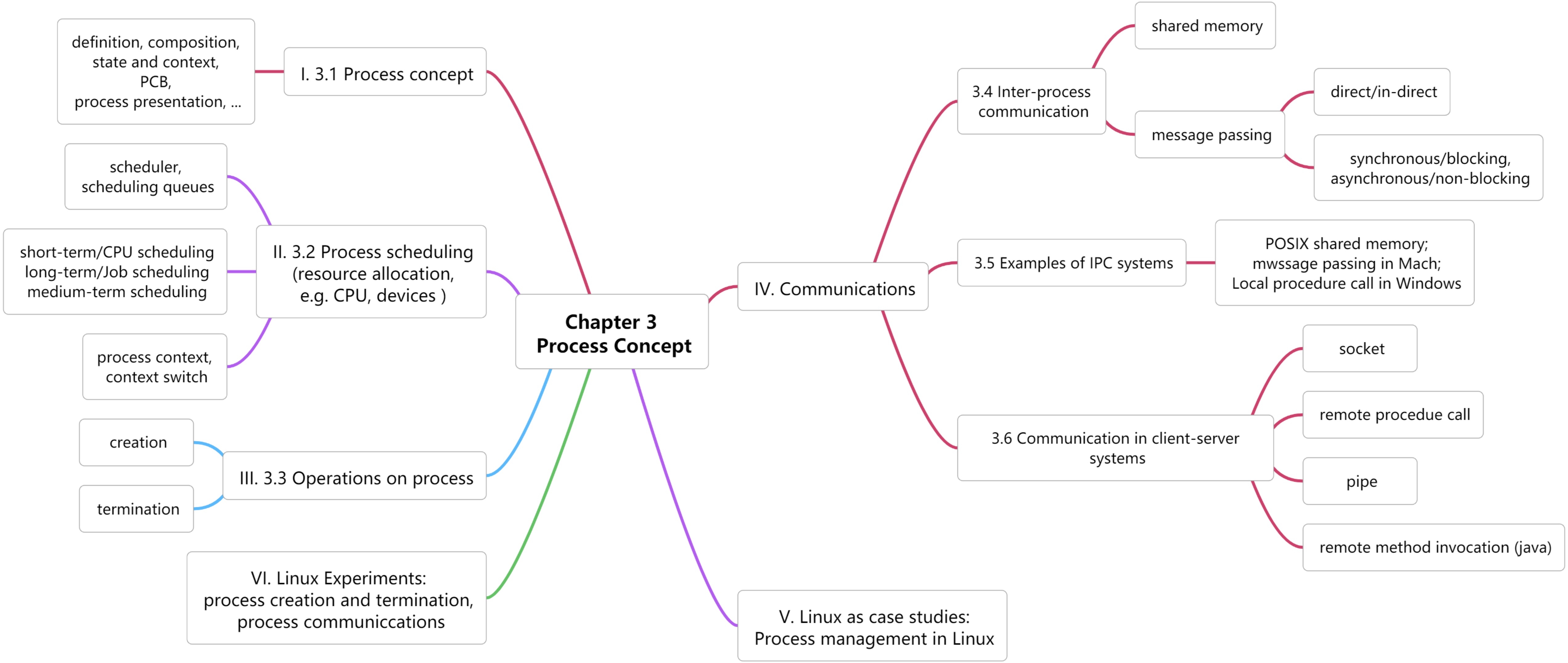
薛 哲

School of Computer Science (National Pilot Software Engineering School)

Part Two Process Management

- **Part Two Process Management**
  - I. Programs in execution
    - Chapter 3 Process Concept (definition, composition, state transition, management, communication)
    - Chapter 4 Multithread Programming
  - II. Chapter 5 Process Scheduling (resource allocation, e.g. CPU, devices)
  - III. Chapter 6 Synchronization (direct/in-direct inter-process interaction)
  - IV. Chapter 7 Deadlocks (resource competition, resource-limitted)

Chapter 3 Process Concept



Outline

- Process Concept

introduce the process -- a program in execution,
which forms the basis of all computation,
its definition, composition, and state transition …

- Process Scheduling
- Operations on Processes

describe process management, including
scheduling(allocating CPU), creation and termination,
and communication

- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

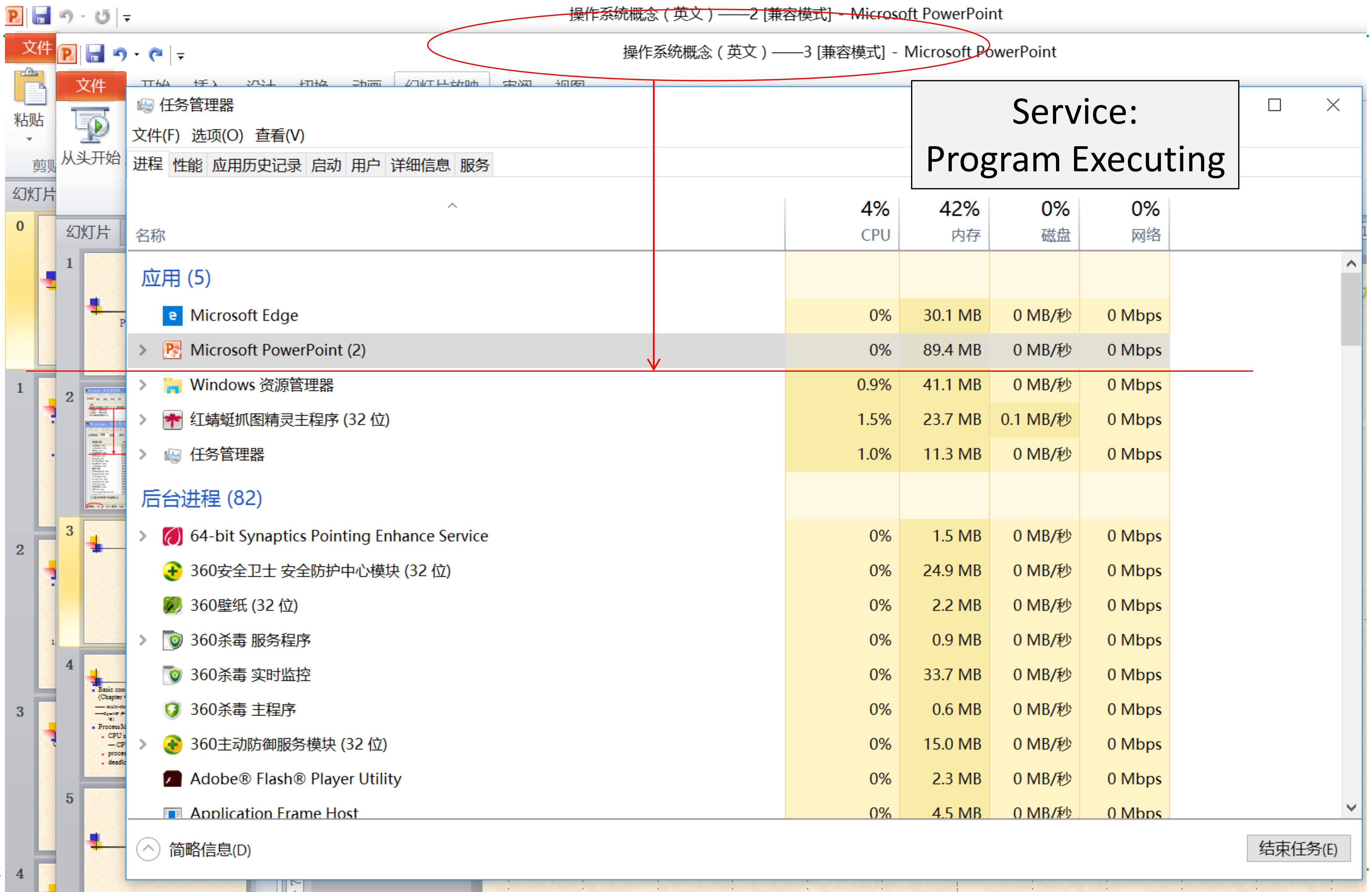
explore interprocess communication using
shared memory and message passing,
and describe communication in client-server
systems

3.1 Process Concept

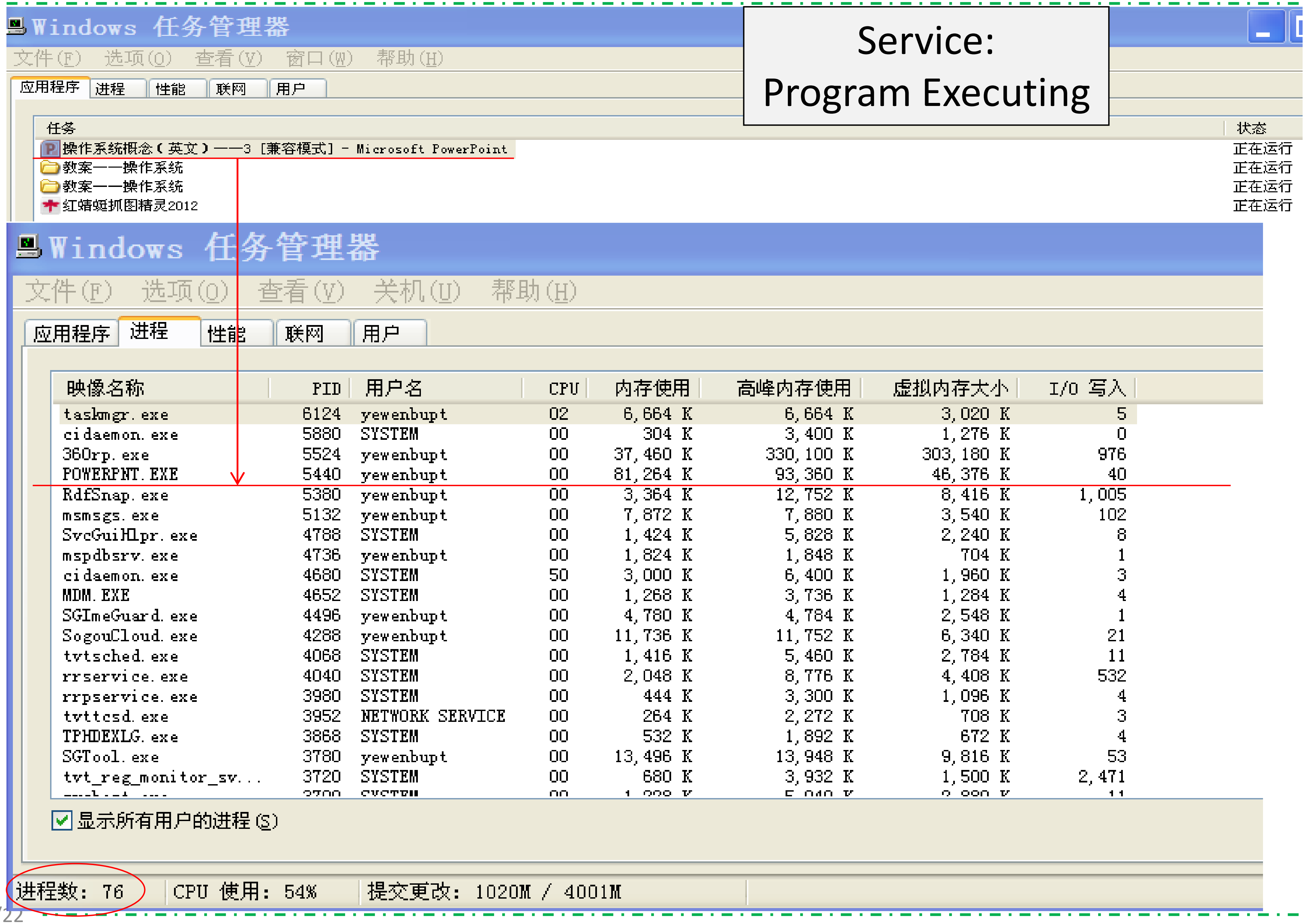
- In multiprogramming systems and time-shared systems
  - more than one programs, i.e. jobs, user programs or OS programs, execute concurrently
    - jobs in batch systems, tasks in time-shared systems
  - one program may execute several times, processing different data in each time
  - different programs' execution phases

    e.g., start, execute, halt, end → process states
- Processes are thus used for describing of concurrent executing of programs
- **Process**
  - a program in execution
  - process execution must progress in sequential fashion, i.e. process states
  - OS allocates CPU, memory, devices to process
- Textbook uses the terms ***job*** and ***process*** almost interchangeably

Service:
Program Executing



Service: Program Executing



映像名称	PID	用户名	CPU	内存使用	高峰内存使用	虚拟内存大小	I/O 写入
taskmgr.exe	6124	yewenbupt	02	6,664 K	6,664 K	3,020 K	5
cidaemon.exe	5880	SYSTEM	00	304 K	3,400 K	1,276 K	0
360rp.exe	5524	yewenbupt	00	37,460 K	330,100 K	303,180 K	976
POWERPNT.EXE	5440	yewenbupt	00	81,264 K	93,360 K	46,376 K	40
RdfSnap.exe	5380	yewenbupt	00	3,364 K	12,752 K	8,416 K	1,005
msmsgs.exe	5132	yewenbupt	00	7,872 K	7,880 K	3,540 K	102
SvcGuiHlpr.exe	4788	SYSTEM	00	1,424 K	5,828 K	2,240 K	8
mspdbsrv.exe	4736	yewenbupt	00	1,824 K	1,848 K	704 K	1
cidaemon.exe	4680	SYSTEM	50	3,000 K	6,400 K	1,960 K	3
MDM.EXE	4652	SYSTEM	00	1,268 K	3,736 K	1,284 K	4
SGImeGuard.exe	4496	yewenbupt	00	4,780 K	4,784 K	2,548 K	1
SogouCloud.exe	4288	yewenbupt	00	11,736 K	11,752 K	6,340 K	21
tvtsched.exe	4068	SYSTEM	00	1,416 K	5,460 K	2,784 K	11
rrservice.exe	4040	SYSTEM	00	2,048 K	8,776 K	4,408 K	532
rrpservice.exe	3980	SYSTEM	00	444 K	3,300 K	1,096 K	4
tvttcsd.exe	3952	NETWORK SERVICE	00	264 K	2,272 K	708 K	3
TPHDEXLG.exe	3868	SYSTEM	00	532 K	1,892 K	672 K	4
SGTool.exe	3780	yewenbupt	00	13,496 K	13,948 K	9,816 K	53
tvt_reg_monitor_sv...	3720	SYSTEM	00	680 K	3,932 K	1,500 K	2,471

- Resources allocated to programs in execution

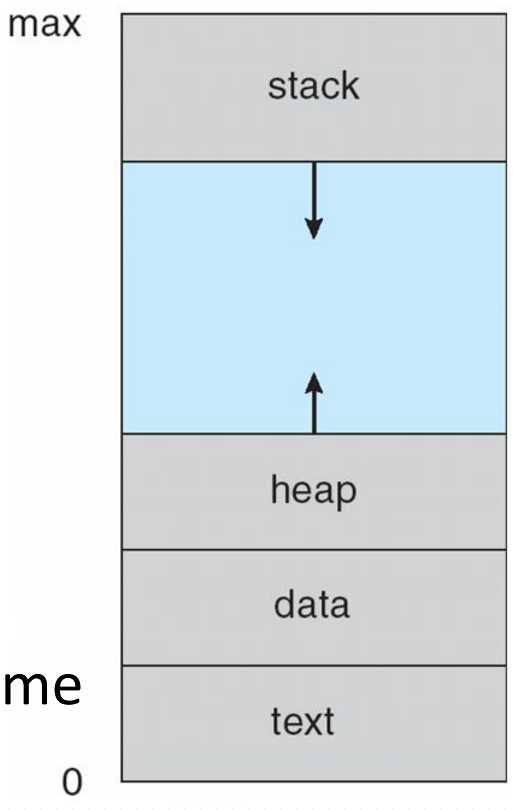
Windows 任务管理器

文件(F) 选项(O) 查看(V) 帮助(H)

应用程序 进程 性能 联网 用户

映像名称	PID	用户名	CPU	CPU 时间	内存使用	高峰内存使用	线...	I/O 读取	I/O 写入	I/O 其他	I/O 读取字节	I/O 写入字节
OSPPSVC.EXE	5928	NETWOR...	00	0:00:05	11,604 K	12,328 K	7	590	483	24,073	516,454	6,664
PopWndLog.exe	5860	yewenbupt	00	0:00:00	5,248 K	5,256 K	2	2	0	874	155,665	0
taskmgr.exe	5768	yewenbupt	05	0:00:09	3,208 K	5,660 K	3	5	5	1,557	340	360
SogouCloud.exe	5760	yewenbupt	00	0:00:04	15,720 K	15,912 K	24	2,577	2,236	188,433	1,642,323	856,397
RdfSnap.exe	4952	yewenbupt	00	0:00:22	9,616 K	12,272 K	3	1,855	1,175	17,457	2,305,544	4,458
msmsgs.exe	4944	yewenbupt	00	0:00:00	7,112 K	7,120 K	2	112	42	2,072	54,474	74,144
360rp.exe	4620	yewenbupt	00	0:00:46	60,352 K	347,336 K	55	81,849	28,635	1,295,279	885,566,197	48,198,231
unsecapp.exe	4492	SYSTEM	00	0:00:00	4,184 K	4,196 K	2	3	2	966	25,280	144
SvcGuiHlpr.exe	4268	SYSTEM	00	0:00:00	5,816 K	5,824 K	2	8	8	1,683	544	576
TPHDEXLG.exe	4032	SYSTEM	00	0:00:00	1,892 K	1,892 K	5	471	459	60,986	22,960	4,436
tvt_reg_monitor_svc.exe	3936	SYSTEM	00	0:00:06	3,644 K	3,988 K	12	2,914	2,912	1,390,249	37,438,814	230,028
svchost.exe	3888	SYSTEM	00	0:00:01	5,060 K	5,228 K	6	453	456	341	42,512	4,168
sqlwriter.exe	3848	SYSTEM	00	0:00:00	3,792 K	3,808 K	3	457	456	794	45,524	4,504
sqlbrowser.exe	3800	NETWOR...	00	0:00:01	8,400 K	8,412 K	12	449	449	1,511	47,918	5,990
RegSrvc.exe	3756	SYSTEM	00	0:00:00	3,392 K	3,400 K	3	446	445	517	42,960	3,684
DkIcon.exe	3640	yewenbupt	00	0:00:00	3,184 K	3,184 K	2	0	0	712	0	0
PanGPS.exe	3588	SYSTEM	00	0:00:01	12,140 K	12,148 K	16	618	548	128,410	524,146	92,462
sqlservr.exe	3424	SYSTEM	00	0:00:06	111,044 K	111,048 K	36	3,668	1,990	15,692	62,640,019	14,912,991
sqlservr.exe	3404	NETWOR...	00	0:00:03	1,976 K	30,308 K	23	948	590	6,241	6,900,449	3,463,624
alg.exe	3344	LOCAL ...	00	0:00:00	3,880 K	3,880 K	6	446	445	881	39,424	3,684
cidaemon.exe	3280	SYSTEM	00	0:00:00	280 K	3,372 K	2	1	0	628	25,144	0
SoftMgrLite.exe	3212	yewenbupt	00	0:00:01	11,320 K	11,336 K	5	91	10	2,209	815,686	2,060
wmiprvse.exe	2776	NETWOR...	00	0:00:28	7,588 K	8,388 K	5	190	52	33,607	769,921	3,935
~~POWERPNT.EXE~~	~~2736~~	~~yewenbupt~~	~~00~~	~~0:01:35~~	~~101,788 K~~	~~138,236 K~~	~~21~~	~~600~~	~~842~~	~~157,785~~	~~24,085,215~~	~~2,519,695~~
conime.exe	2520	yewenbupt	00	0:00:00	3,112 K	3,112 K	1	0	0	557	0	0
ApntEx.exe	2384	yewenbupt	00	0:00:00	3,224 K	3,340 K	2	0	0	1,471	0	0
ApMsgFwd.exe	2244	yewenbupt	00	0:00:05	2,264 K	2,264 K	3	0	0	486	0	0
AcSvc.exe	2216	SYSTEM	00	0:00:41	19,500 K	19,592 K	22	620	506	65,924	588,168	7,484

Process Concept



- A process consists of
  - The program code, also called **text section**（代码段）
  - Current activity including **program counter**, processor registers
    - indicating process execution traces
  - **Stack**（栈区） containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section**（数据段） containing global variables
  - **Heap**（堆区） containing memory dynamically allocated during run time
- Program vs process
  - Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
    - Program becomes process when executable file loaded into memory
  - 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 State

- A process is dynamic, and has its *lifetime.*
- 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

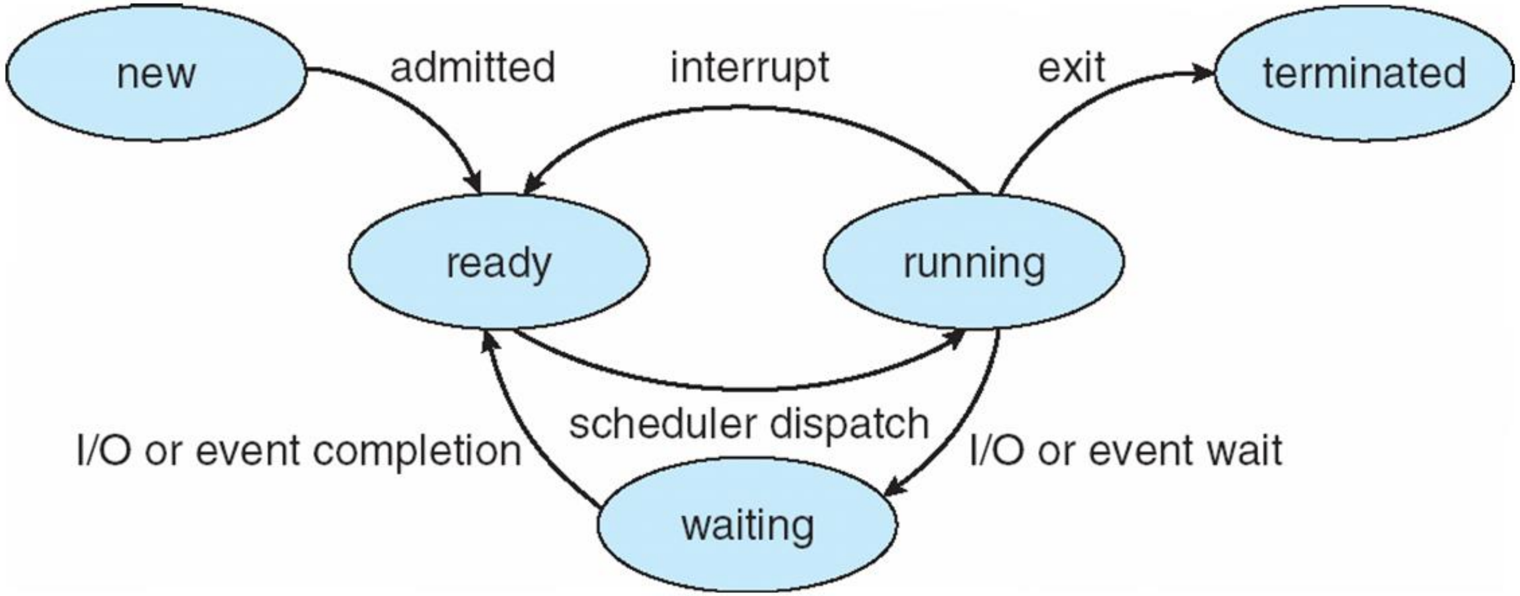
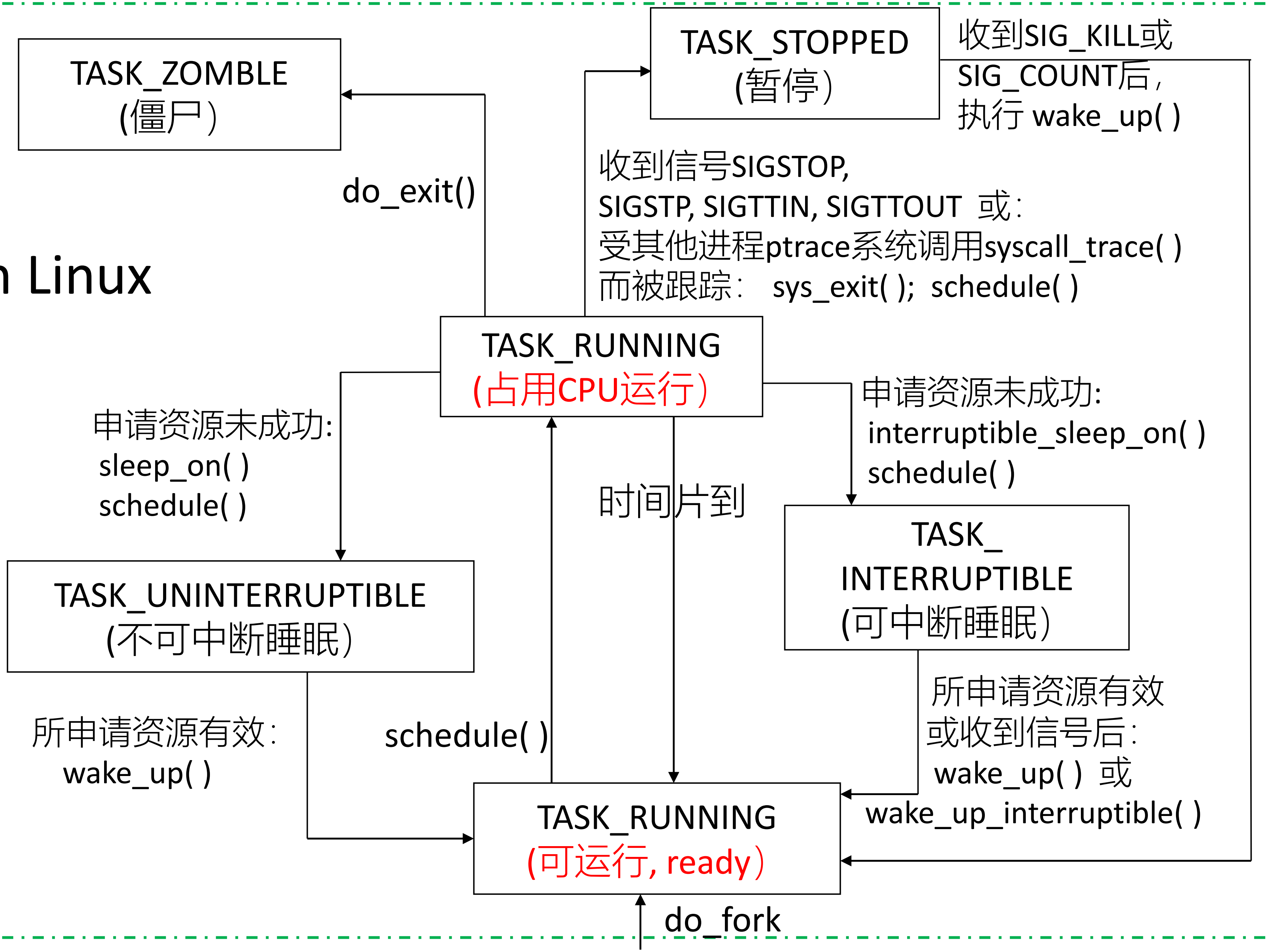


Fig. 3.2 Process States

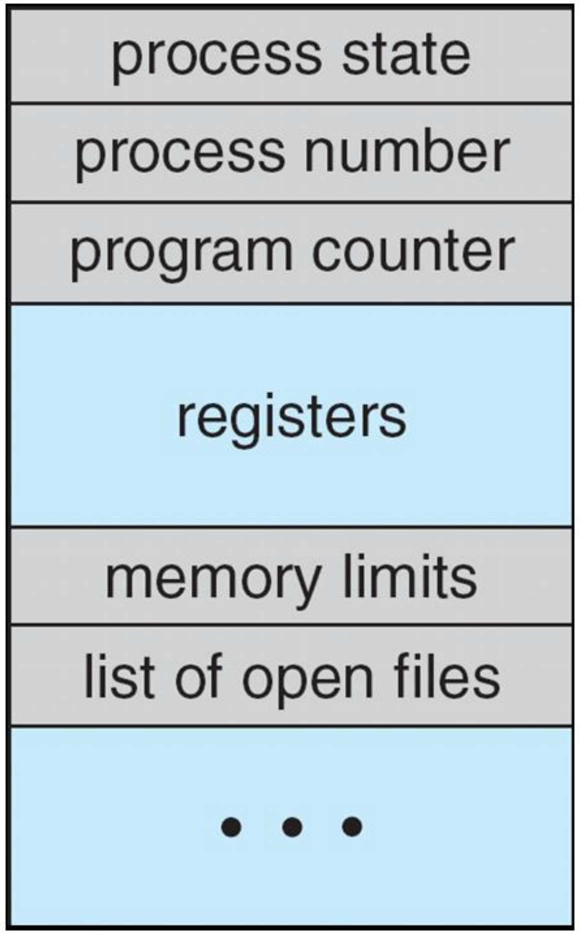
A practical OS such as Linux, may have different states from those in Fig.3.2

States in Linux



Process Control Block (PCB)

- PCB
  - a data structure in kernel used for OS to manage process
  - also called **task control block**
- Information associated with each process in PCB
  - 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



Process context

Process State

- A Exercise
  - which of the following information are not contained in PCB ( )

    A. Process state　　　　B. Program counter

    C. User data　　　　D. CPU registers

  - answer: C

CPU Switch From Process to Process

- Several processes alternatively occupies and runs on the CPU

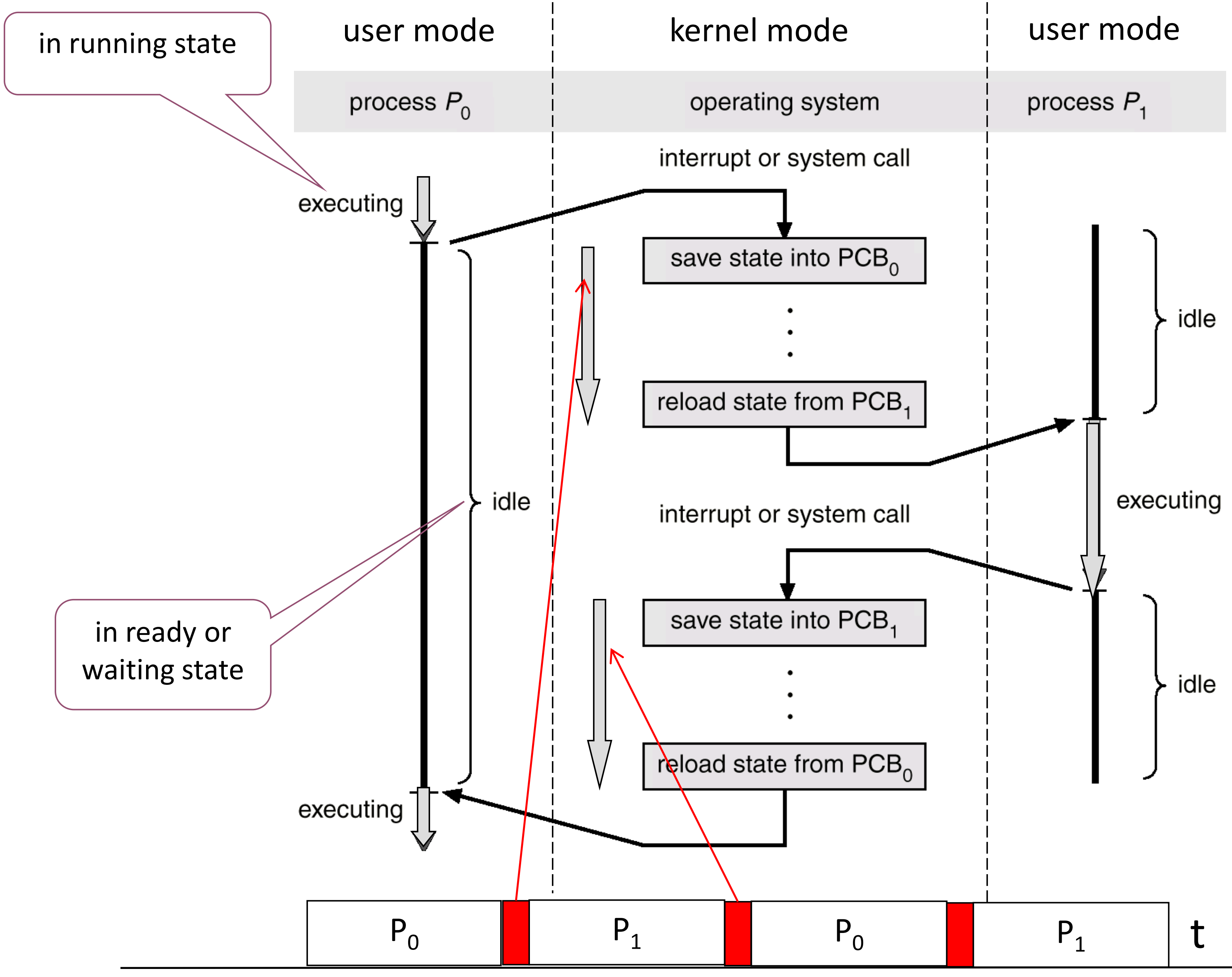


Fig. 3.4 ***CPU Switch*** From Process to Process

Threads

- So far, process has a single thread of execution
- Consider multi-thread programming, 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
- See next chapter

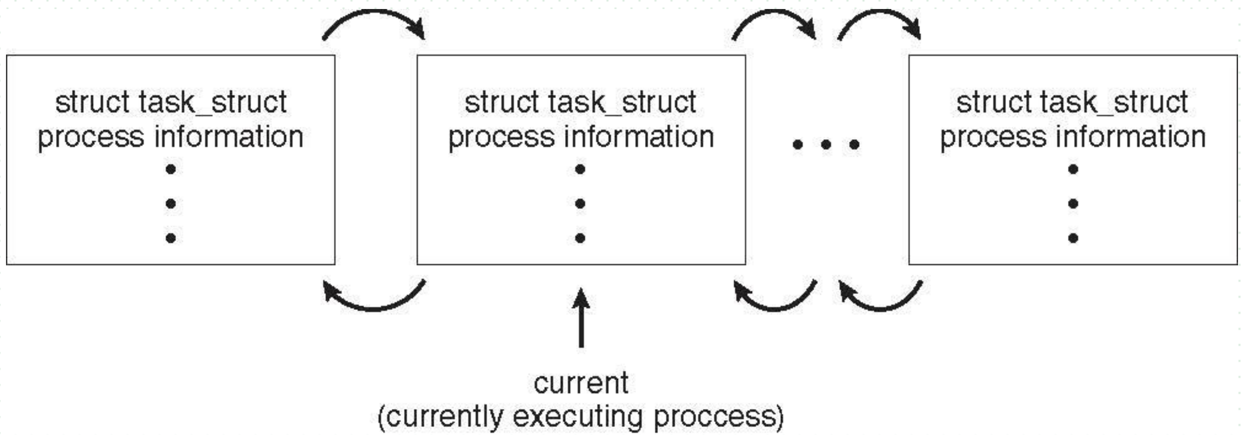
Process Representation in Linux

- Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

Linux
的task_struc

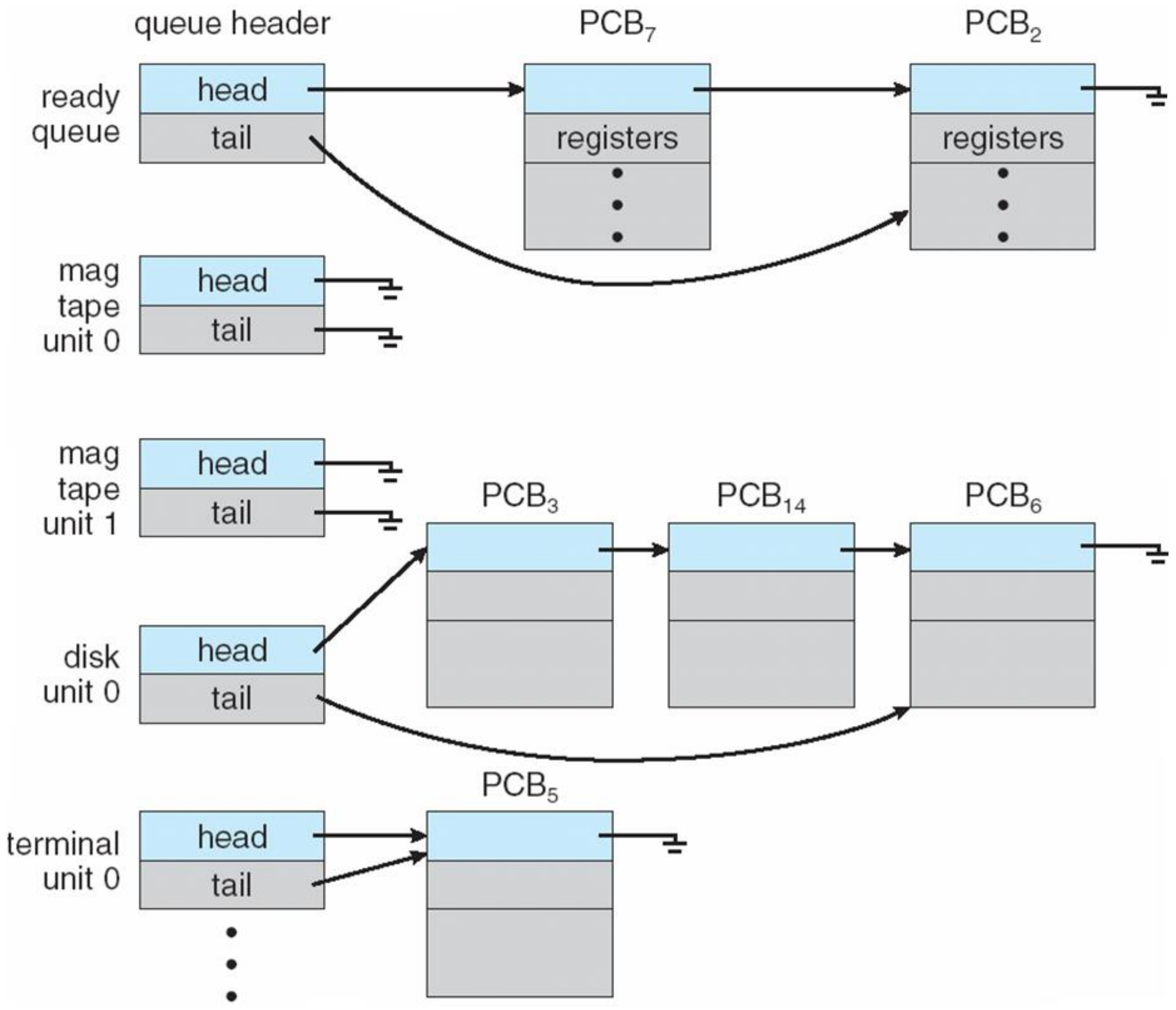
PCB in Linux, task_struc



3.2 Process Scheduling

- Selecting processes and allocating CPU, devices etc. resources to them
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue and Various I/O Device Queues



Process Migrates between The Various Queues

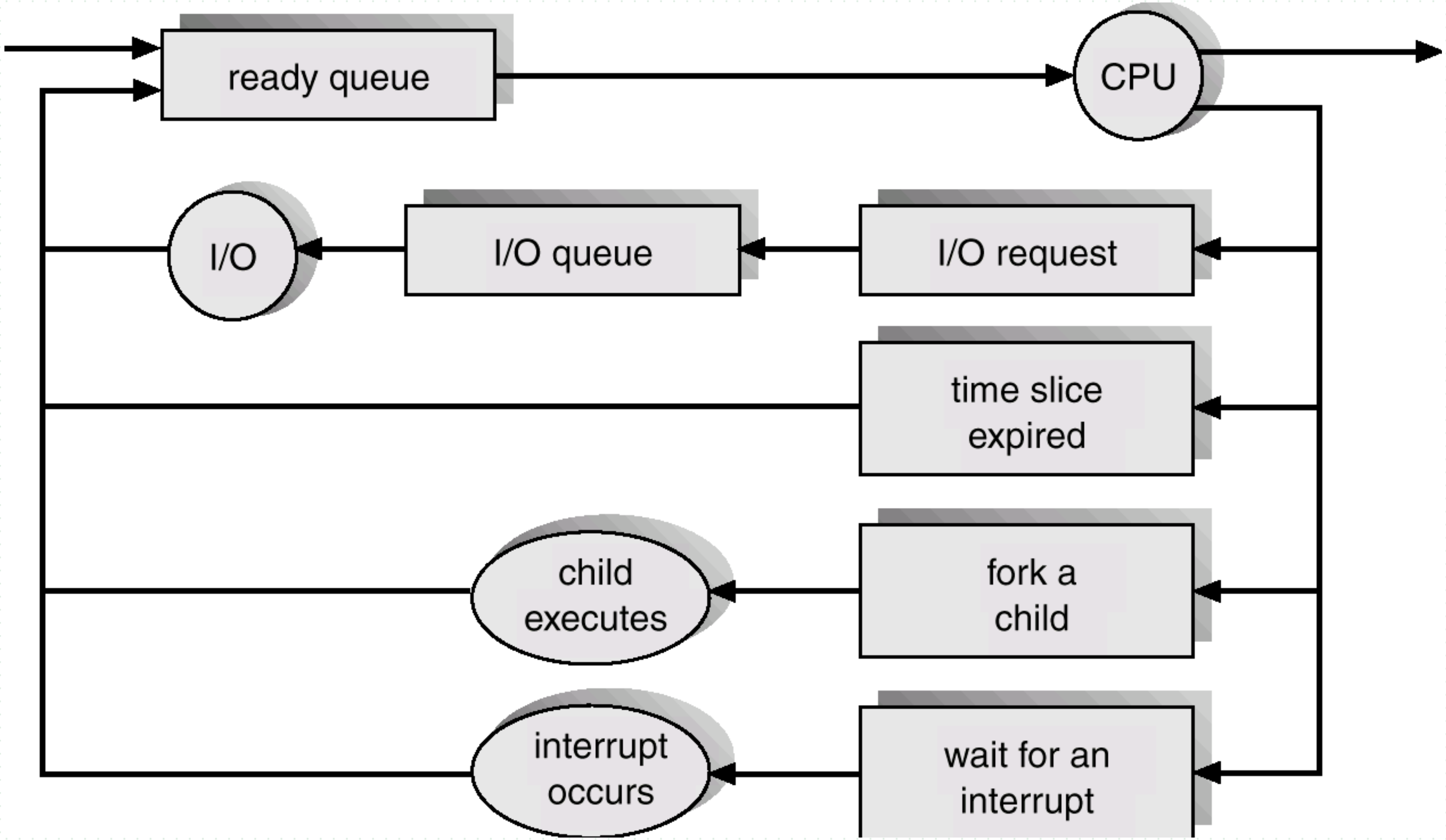


Fig. 3.7 Representation of Process Scheduling

Schedulers

- Degree of multiprogramming (多道程序设计粒度)
  - the number of process in (main) memory
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- In batch systems (e.g. mainframe systems), more processes (or user jobs) are submitted to the system than can be executed immediately. These processes/jobs are spooled ***as jobs*** to a mass-storage device (typical a disk) , where they are kept for later execution
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**

long-term scheduling(LTS) vs short-term scheduling(STS)

Noj > MMPD!

total number of jobs submitted by users(NoJ)

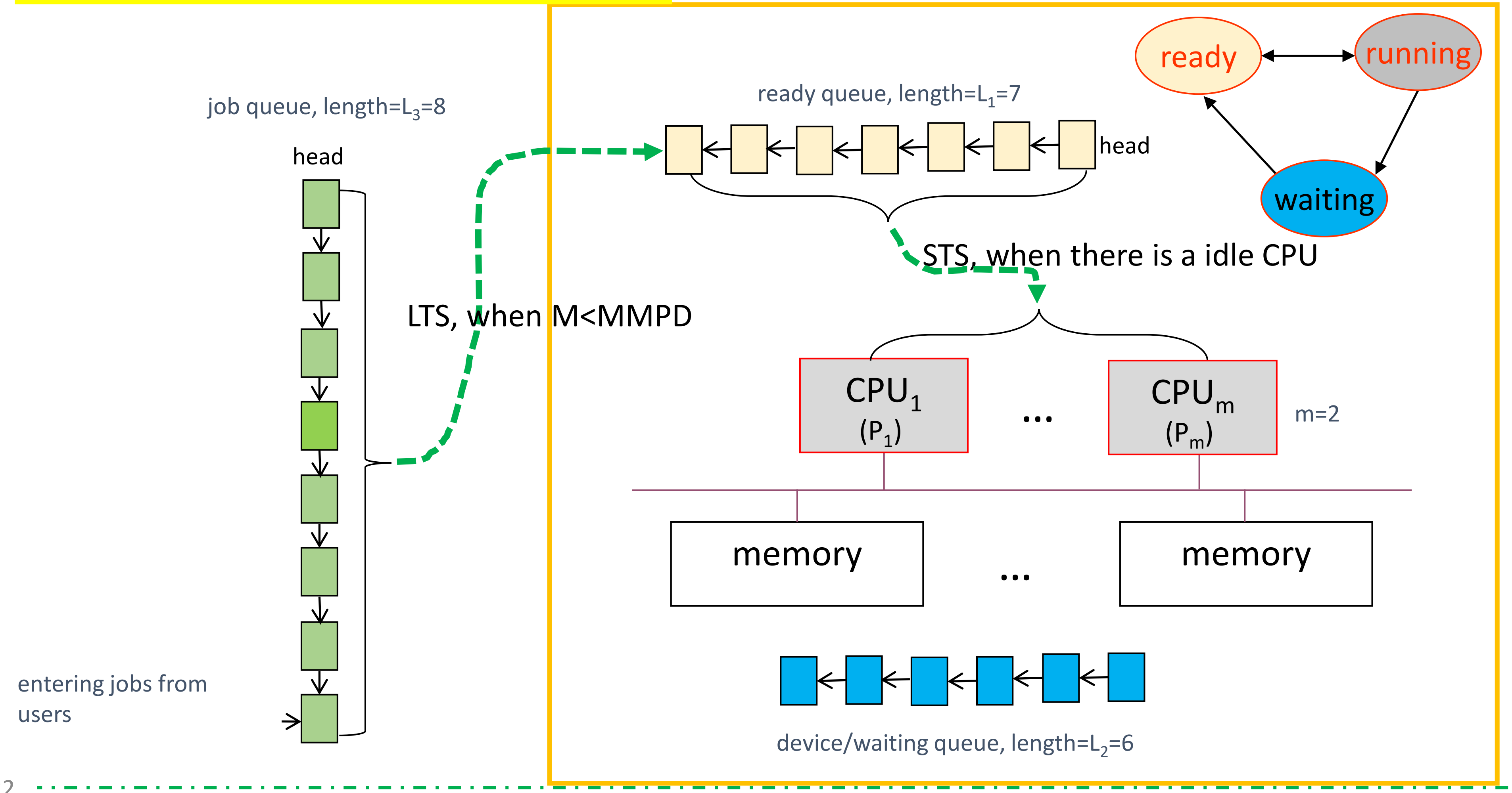
$Noj=L_3 + M = 8+15=23$

Maximum multiple programming degree(MMPD)=16

M concurrent processes in memory

M=m + L1 + L2=2+7+6=15

要求：$M \leq MMPD$

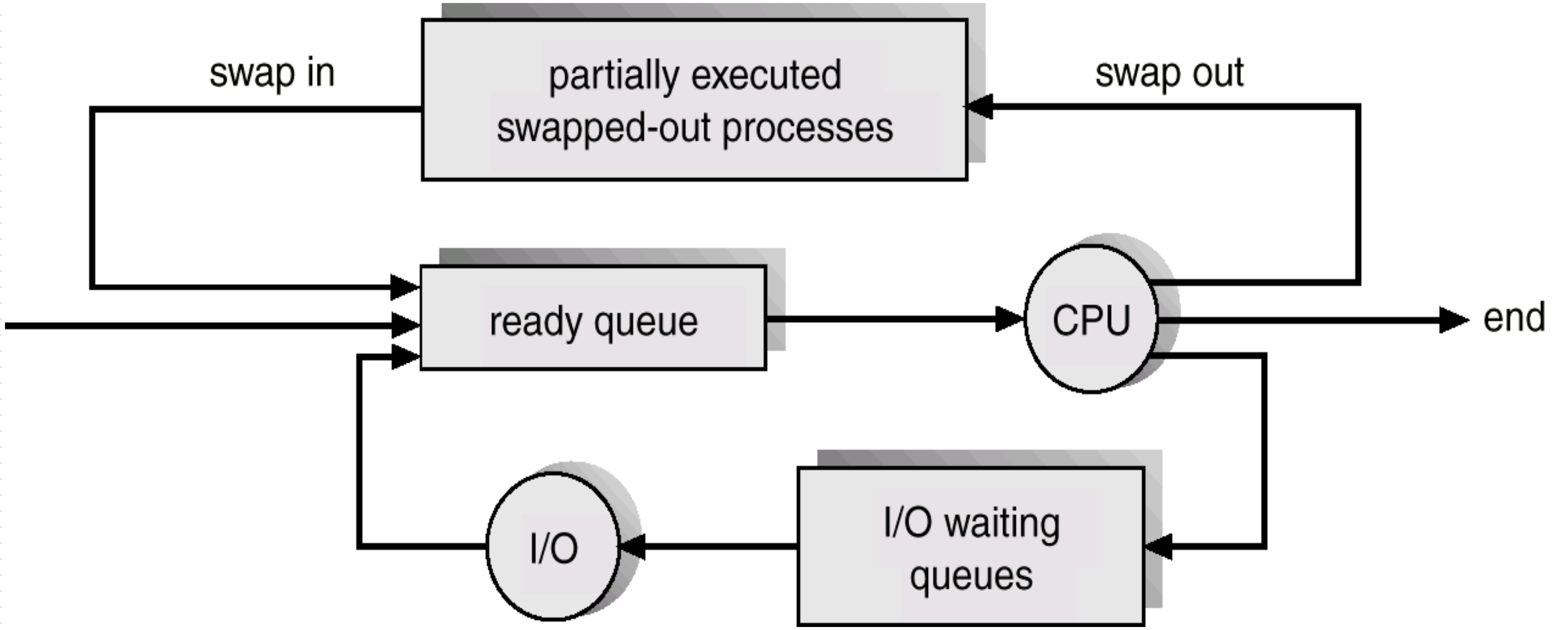


Schedulers

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts, also called I/O-intensive process
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts, also called computation-intensive process
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- In some operating systems, such as time-sharing systems, medium term scheduling exists
  - to control the degree of multiprogramming, guaranteeing the resources (such as main memory) demanded by all processes in memory has not overcommitted available resources
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

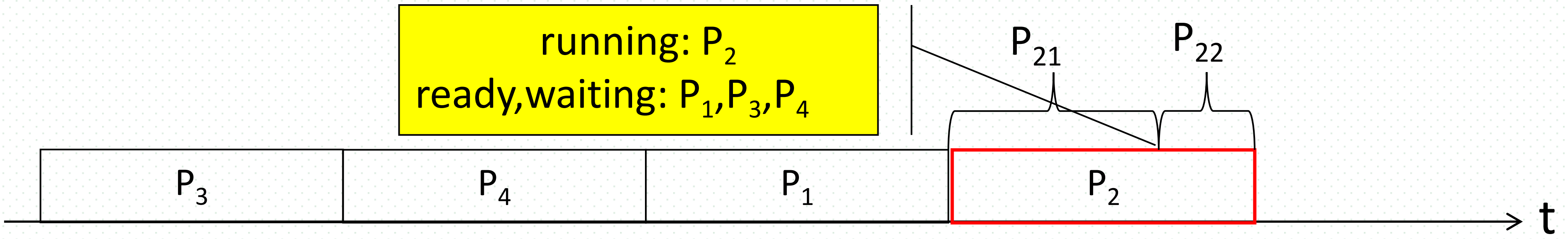


Example

- Four processes P_1, P_2, P_3 and P_4 concurrently run in main memory
  - P_1, P_2, P_3 and P_4 are 300K, 700K, 800K, and 700K total in size respectively, 300+700+800+700=2500
  - main memory is divided into five sections, i.e, ***section0,*** ***section*1*, section*2*, section*3**, and ***section*4**, allocated to **OS** and the four ***processes*** to be loaded to execute
  - the size of each section is 200K, 200K, 300K, 600K, 200K respectively, and is smaller than the size of the process running in it
    - 200 + 200 + 300 + 600 + 200 = 1500 < 2500

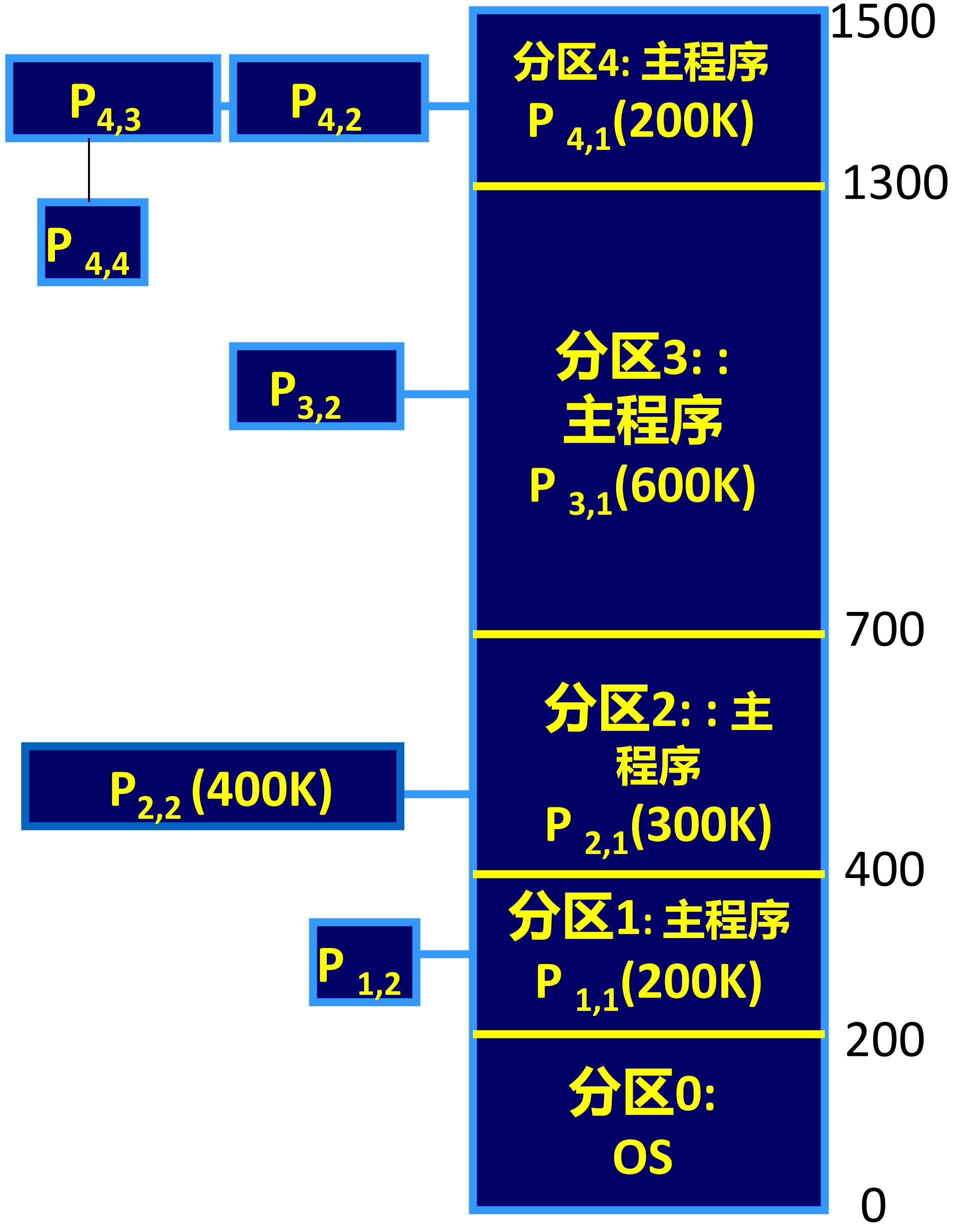
Example

- each process is divided into several parts, and each part is loaded into memory to run *sequentially*
  - P_1(300K) = {main program P_{11}(200K) , subroutine P_{12}(100K)},
  - P_2 (700K) ={main program P_{21} (300K), subroutine P_{22} (400K) }
  - P_3 (800K) ={main program P_{31} (600K), function P_{32} (200K) }
  - P_4 (700K) ={main program P_{41} (200K), procedure P_{42} (200K), P_{43} (200K), P_{44} (100K)}
- initially, each process executes its first part e.g. main program, in the section allocated to it, as described in Fig.4.0.2
- when the subroutine $P_{2,2}$ (400K) are loaded to memory, the main program $P_{3,1}$ (600K) , which is still in execution, is swapped out

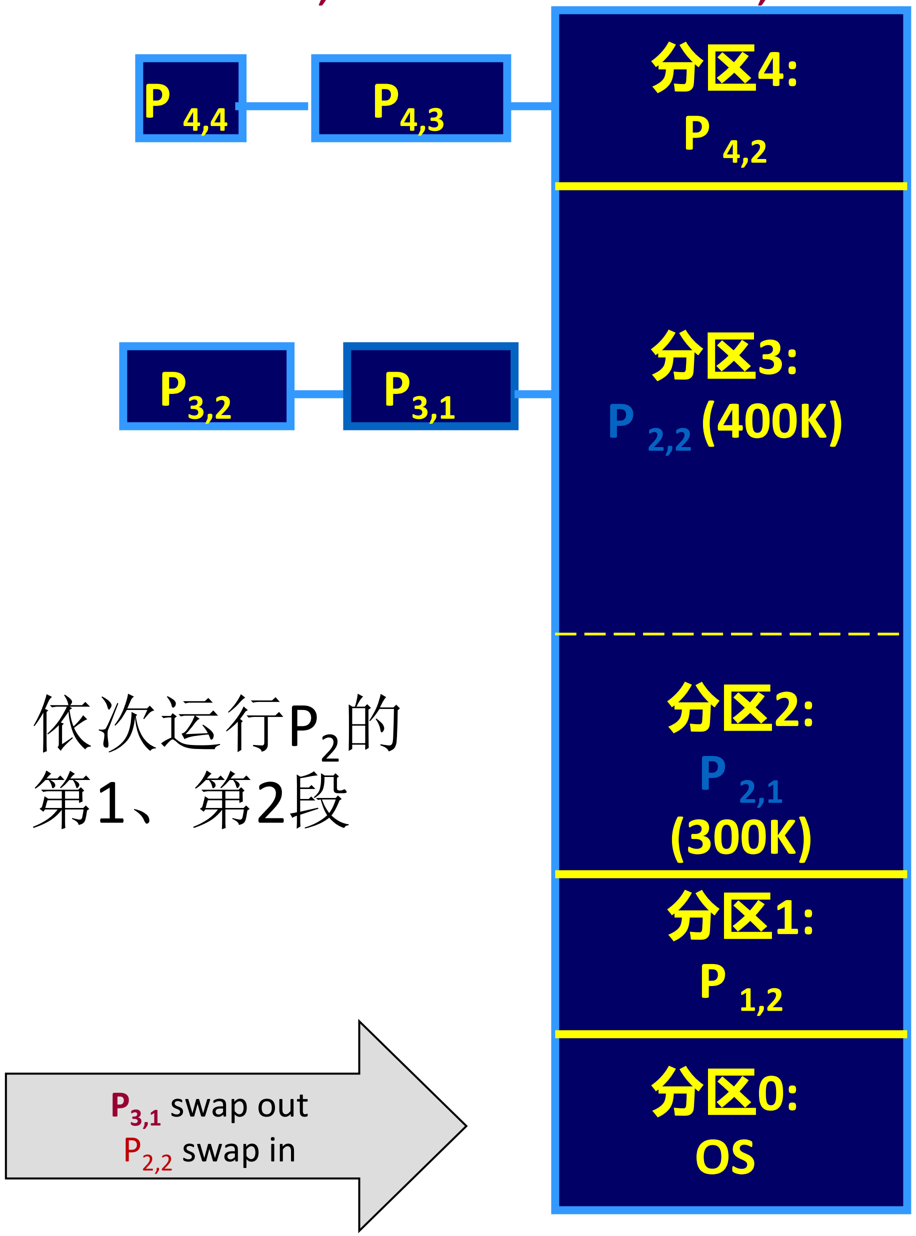


time-sharing scheduling

各个进程运行其第一段主程序

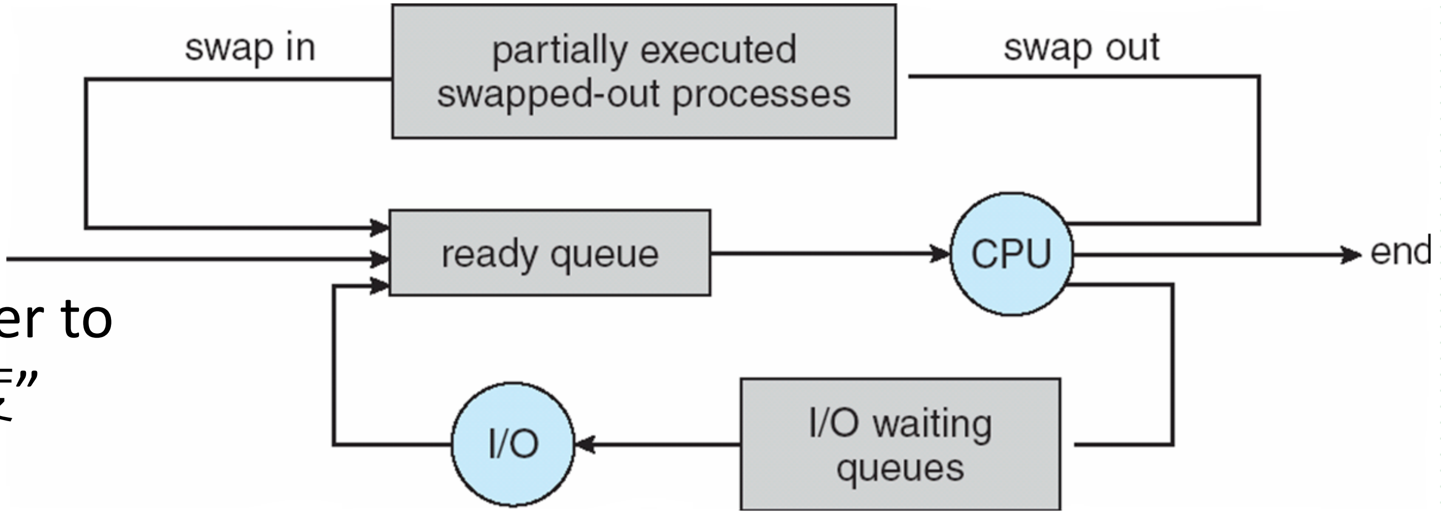


将子程序$P_{2,2}$调入内存时, $P_{3,1}$换出



Addition of Medium Term Scheduling

- Medium term scheduling is responsible for swapping (交换/倒换)
  - swapping out
    - to reduce degree of multiprogramming and free memory or resources, removing processes in ready or running states from main memory and into *swapping section* on disks
    - the process that is swapped out is delayed to execute
  - swapping in
    - reintroducing the processes on the disk into memory into memory, restarting their executing



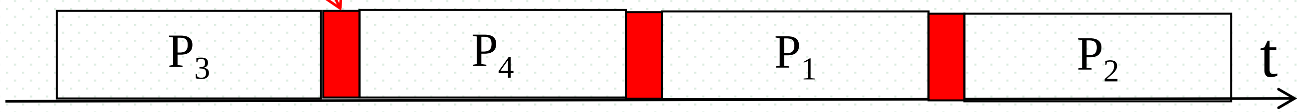
- For more details about scheduling, refer to “Appendix 3.A调度的层次与作业调度”

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

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, e.g.
  - values of the CPU registers, counter
  - the process states
  - memory management information etc., e.g. addrsss space the process located in
- 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



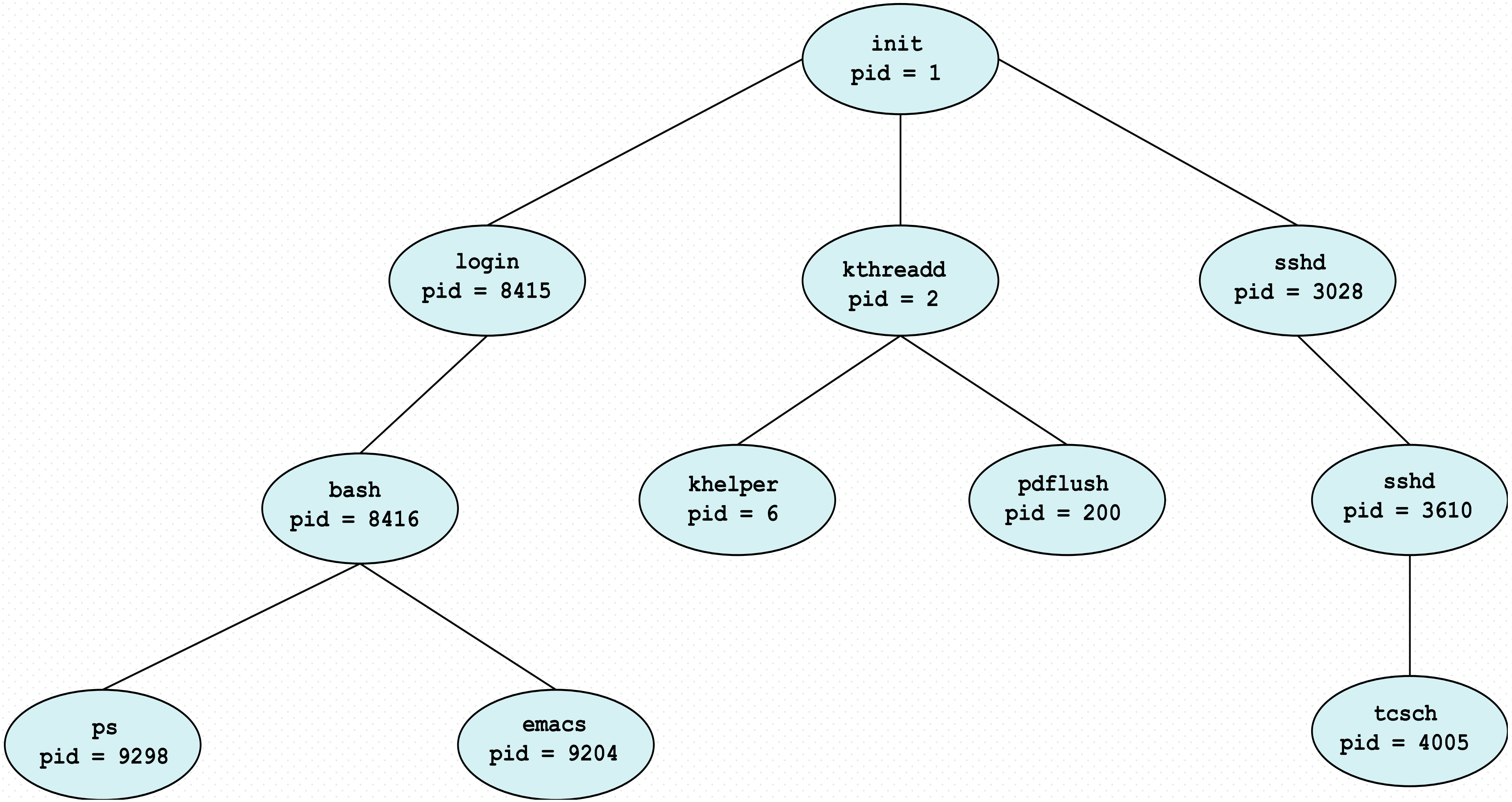
3.3 Operations on Processes

- OS must provide mechanisms, by system calls, for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- OS and other processes use ***creation*** primitive /system call to create a new process
- Actions taken by OS when creating a process include
  - load the program code into main memory allocated to this process
  - allocate ***resources*** (memory, I/O devices, files) to the process
  - **build the PCB for this process**
  - insert the PCB into ready queue, (and the process change into **ready** state)
- The process being created is in ***new*** state
- **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

A Tree of Processes in Linux



任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

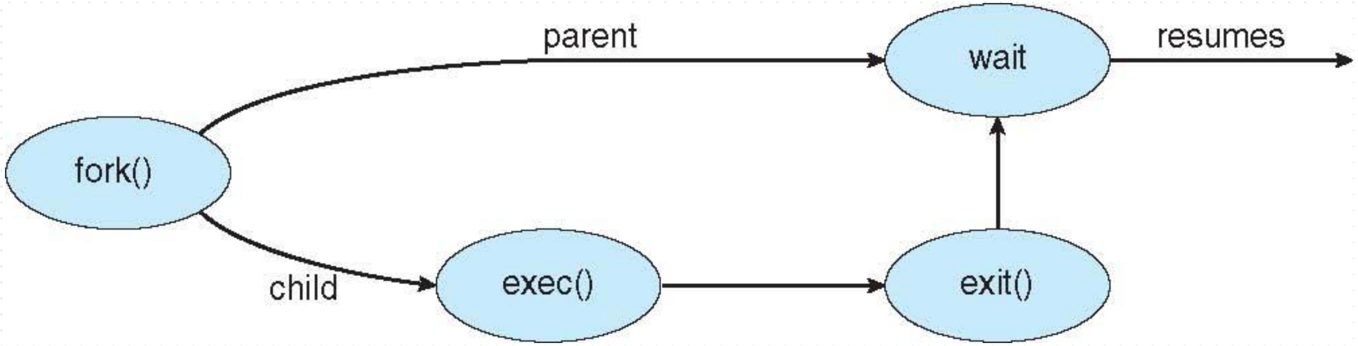
名称	PID	状态	用户名	CPU	内存(活动的专用工作集)	UAC 虚拟化
unsecapp.exe	7856	正在运行	Thinkpad	00	1,120 K	不允许
Video.UI.exe	1608	已挂起	Thinkpad	00	K	不允许
vpnclient_x64.exe	4284	正在运行	SYSTEM	00	5,588 K	不允许
VPNService.exe	4576	正在运行	SYSTEM	00	452 K	不允许
WeChat.exe	1816	正在运行	Thinkpad	00	31,880 K	不允许
WeChatWeb.exe	9792	正在运行	Thinkpad	00	16,072 K	不允许
WeChatWeb.exe	7184	正在运行	Thinkpad	00	41,836 K	不允许
wfcrun32.exe	10440	正在运行	Thinkpad	00	1,608 K	不允许
WindowsInternal.C...	3324	正在运行	Thinkpad	00	3,376 K	不允许
wininit.exe	648	正在运行	SYSTEM	00	336 K	不允许
winlogon.exe	428	正在运行	SYSTEM	00	836 K	不允许
WinStore.App.exe	10932	已挂起	Thinkpad	00	K	不允许
wlanext.exe	3504	正在运行	SYSTEM	00	1,080 K	不允许
WmiPrvSE.exe	5964	正在运行	SYSTEM	00	1,424 K	不允许
WmiPrvSE.exe	6600	正在运行	NETWORK...	00	4,928 K	不允许
WUDFHost.exe	780	正在运行	LOCAL SE...	00	828 K	不允许
WUDFHost.exe	1320	正在运行	LOCAL SE...	00	900 K	不允许
YourPhone.exe	676	已挂起	Thinkpad	00	K	不允许
ZeroConfigService.e...	4532	正在运行	SYSTEM	00	1,704 K	不允许
ZhuDongFangYu.exe	2952	正在运行	SYSTEM	00	3,948 K	不允许
系统中断	-	正在运行	SYSTEM	02	K	
系统空闲进程	0	正在运行	SYSTEM	86	8 K	

简略信息(D) 结束任务(E)

- 内核初始化init模块
  - 加载启动其它内核模块，自身变为0号idle进程
  - 为系统内其他进程的祖先节点

Process Creation

- Address space
  - total range of memory locations addressable by the process
- Two modes for Address space of child process
  - Child duplicate of parent, so parent and child processes can communicate easily, for example, by memory-sharing scheme
    - E.g. Linux/Unix
  - Child has a program loaded into it, i.e., the child has its independent address space
- UNIX/Linux examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process′ memory space with a new program



C Program Forking Separate Process

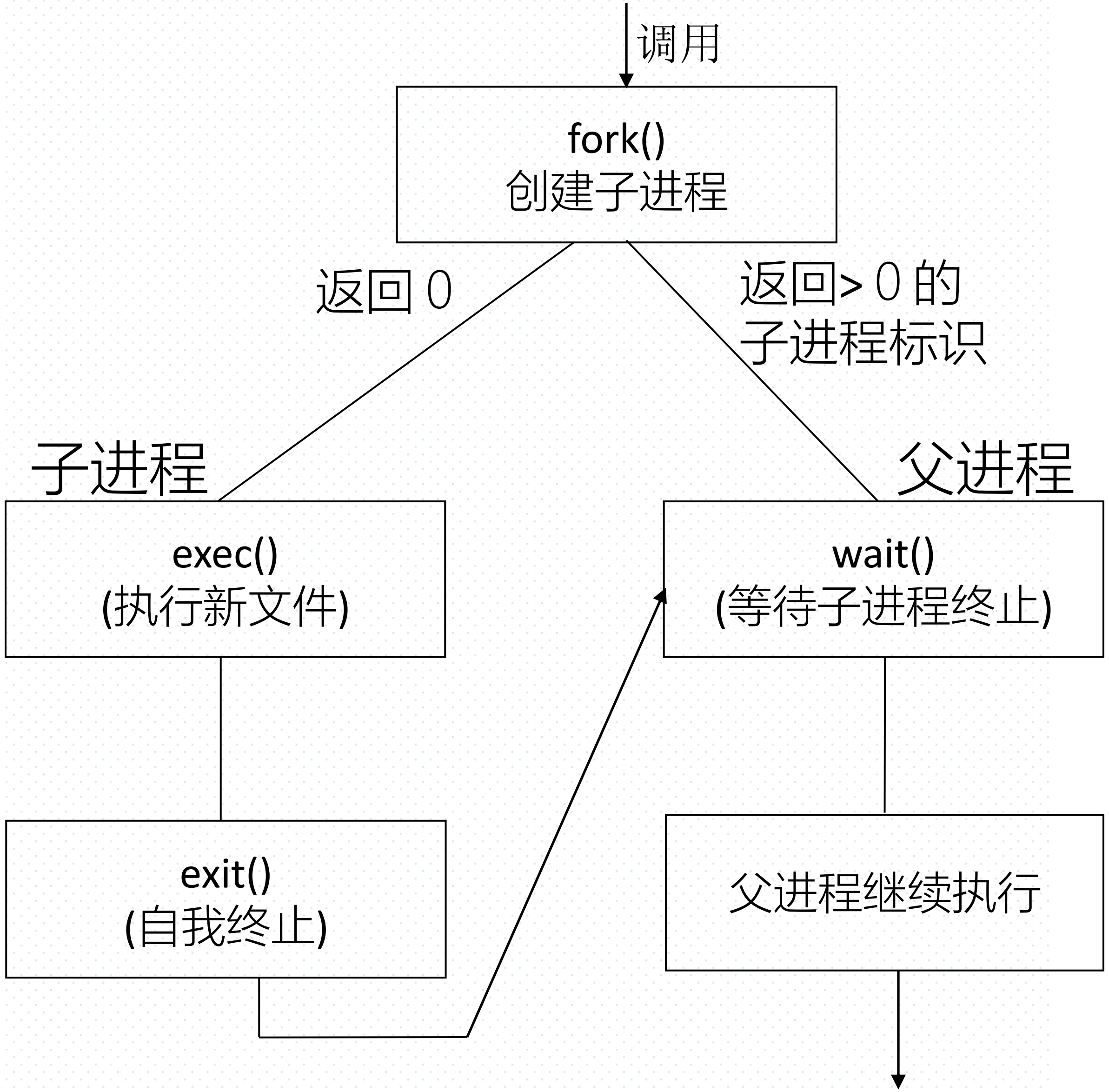


Fig. fork进程创建

```c
#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;
}
```

- E.g. 键盘输入程序(fork, exec, wait系统调用)

```
while {
  显示命令提示符;
  等待用户命令键入命令;
  接收并分析命令行;
  if  (pid=fork())>0
     if (无&)  wait(pid);
  else
     exec( 程序名， 参数)
```

调用；返回；赋值；判断

fork()成功，返回子进程pid>0

父进程

```
while {
  显示命令提示符;
  等待用户命令键入命令;
  接收并分析命令行;
  if  (pid=fork())>0
     if (无&)  wait(pid);
  else
     exec( 程序名， 参数)
```

返回；赋值；判断

fork()创建的子进程

Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- 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

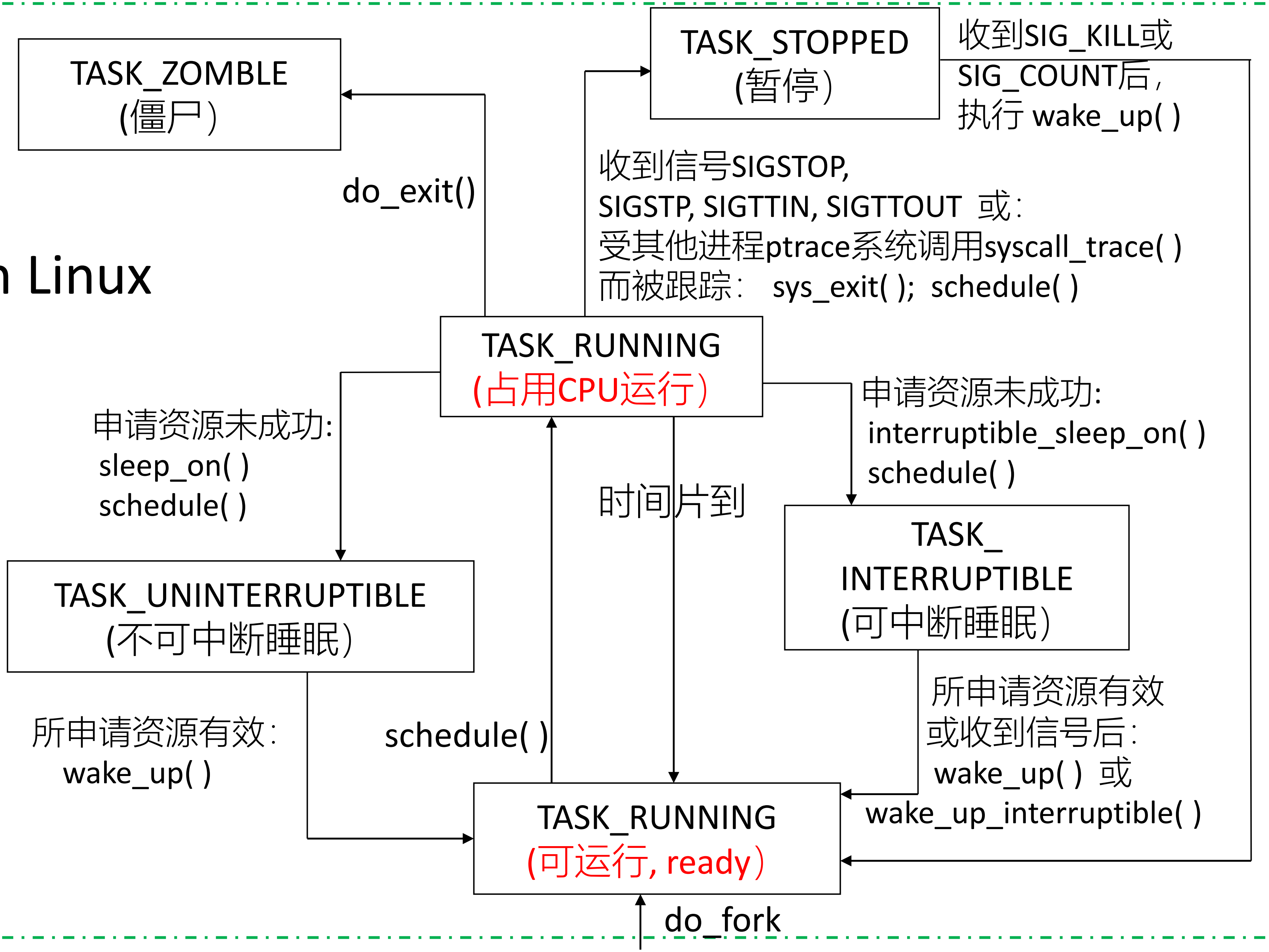
Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. 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, enabling the parent to know which child is terminated

```
pid = wait(&status)
```

- **Zombie process（僵尸）**
  - child is terminated by exit() and its resource is released, but no parent invoke wait, child is at the Zombie state
- **Orphan process（孤儿）**
  - If parent terminated without invoking **`wait`**, the child becomes a orphan
  - init is then taken as its parent

States in Linux



Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** (渲染) process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

3.4 Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory [in user mode]**
  - **Message passing [in kernel mode]**

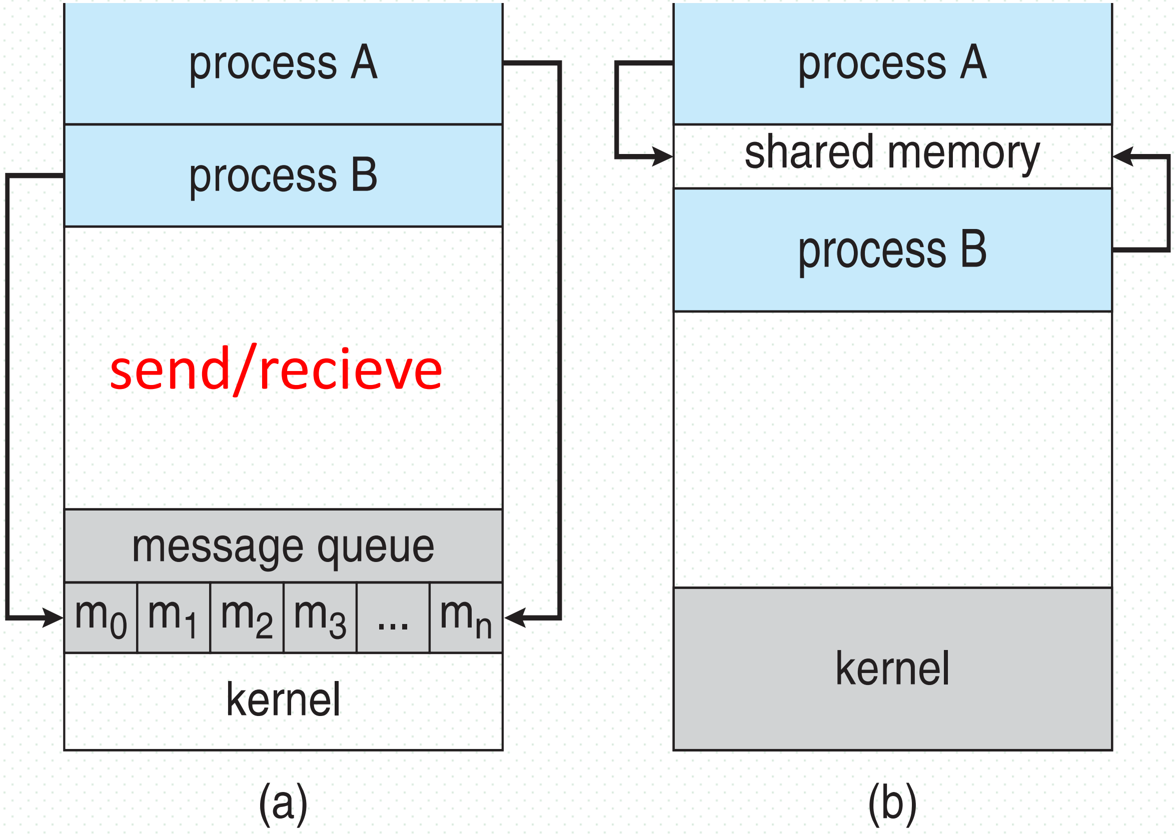
Communications Models

int msgget(key_t, key, int msgflg)　　/创建消息队列

int msgsend(int msgid, const void *msg_ptr, size_t msg_sz, int msgflg)　　/向消息队列发送消息

int msgrcv(int msgid, void *msg_ptr, size_t msg_st, long int msgtype, int msgflg) /从消息队列中取消息

(a) Message passing　　(b) shared memory

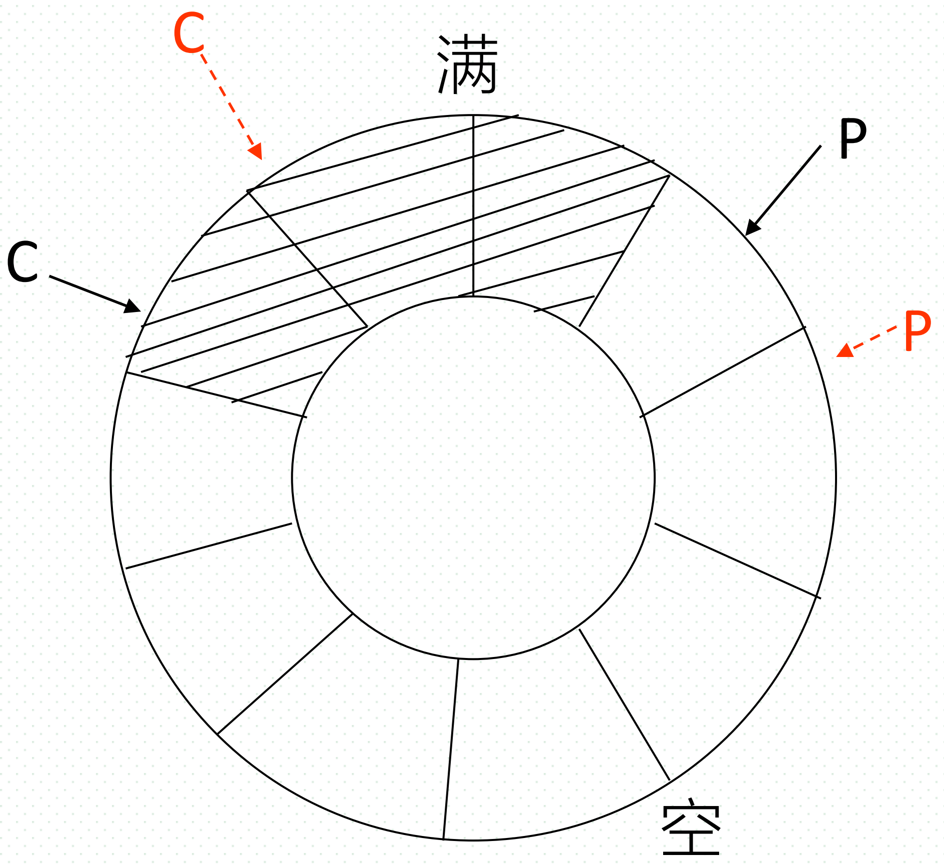


- Concurrent executing of cooperating processes requires OS to provide mechanisms allowing processes
  - to ***communicate*** with one another (§3.4)
  - to ***synchronize*** their actions (chapter 6)

Producer-Consumer Problem

- Paradigm for cooperating processes, in which *producer* process produces information that is consumed by a *consumer* process, through a sharing **buffer**

1. 容量为N的环形缓冲区
 满缓冲区头指针C
 空缓冲区头指针P
2. *n*个生产者， *m*个消费者.
多个生产者和消费者并发运行
3. 生产者—产生数据；
 写入指针P指
 向的空缓冲区；
 指针P前移。
4. 消费者—从指针C指向的满
 缓冲区中取数据；
 指针C前移



Producer-Consumer Problem

- Paradigm for cooperating processes, in which *producer* process produces information that is consumed by a *consumer* process, through a sharing **buffer**
- Cooperating
  - 生产者写数据时有空缓冲块，如果不满足，则阻塞
  - 2个生产者不能同时向同一 空缓冲块写数据
  - 消费者取数据时，有满缓冲块，如果不满足，则阻塞
  - 2个消费者不能同时从同一 满缓冲块取数据
  - 生产者和消费者不能同时对同一缓冲块进行读写操作

- 许多实际问题可抽象为生产者-消费者问题/模型， 如基于邮箱的进程通信, 共享内存通信方式

Producer-Consumer Problem

- Paradigm for cooperating processes, in which *producer* process produces information that is consumed by a *consumer* process, through a sharing **buffer**

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

```
int msgget(key_t, key, int msgflg)                                              /创建消息队列
int msgsend(int msgid, const void *msg_ptr, size_t msg_sz, int msgflg)          /向消息队列发送消息
int msgrcv(int msgid, void *msg_ptr, size_t msg_st, long int msgtype, int msgflg)  /从消息队列中取消息
```

- The *message* size is either fixed or variable

Message Passing (Cont.)

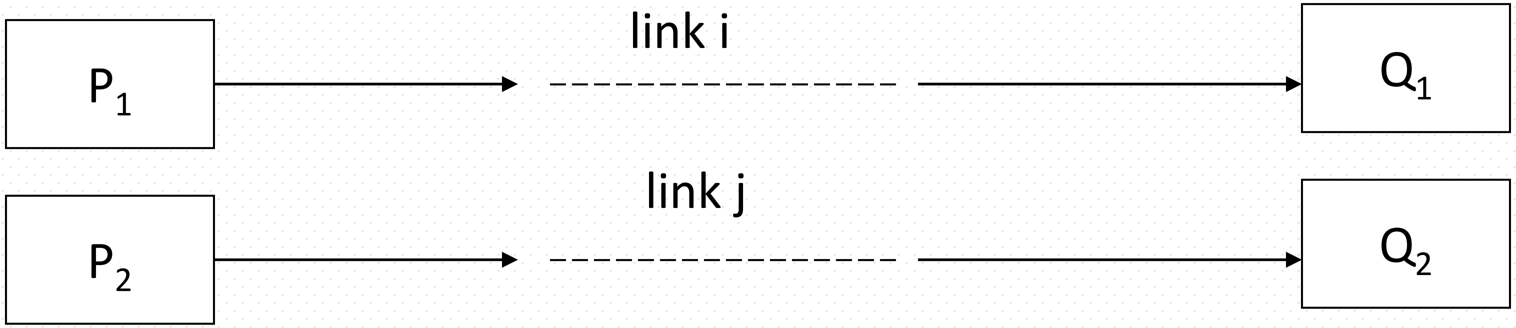
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

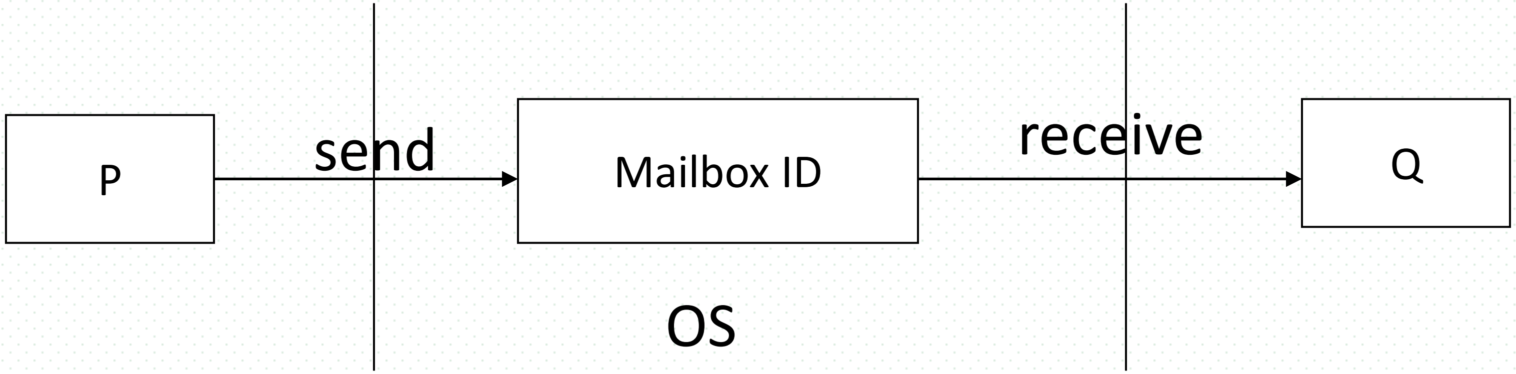
Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*Q, message*) – send a message to process Q
  - **`receive`**(*P, message*) – receive a message from process P
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional



Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id /消息队列
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional



Indirect Communication

- Communication between P and Q
  - P, or Q, or other process ***creates*** a new mailbox ***A***
  - P and Q communicate via ***send***(*mail_**A**, message*) and **receive**(*mail_**A**, message*)
  - when communication is completed, mailbox ***A*** is destroyed
- Primitives are defined as:

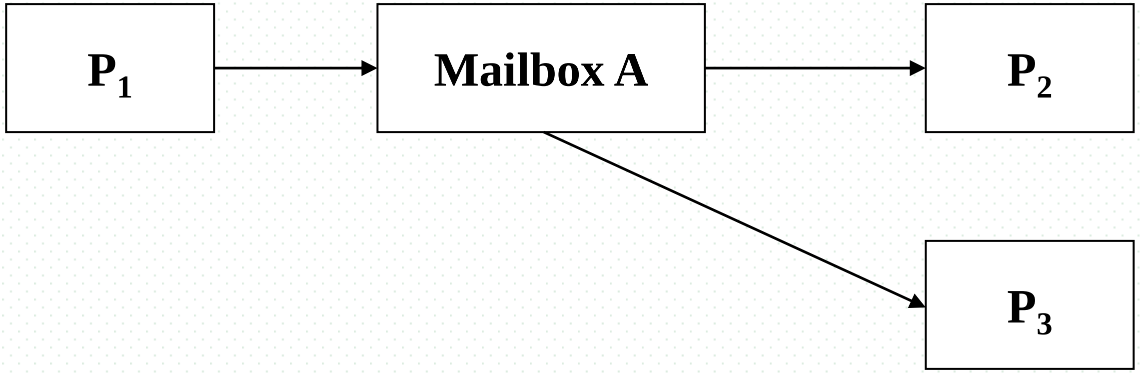
  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

```
int msgget(key_t, key, int msgflg)                                     /创建消息队列
int msgsend(int msgid, const void *msg_ptr, size_t msg_sz, int msgflg)  /向消息队列发送消息
int msgrcv(int msgid, void *msg_ptr, size_t msg_st, long int msgtype, int msgflg)  /从消息队列中取消息
```

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?



- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- **Different combinations possible**
  - If both send and receive are blocking, we have a **rendezvous**

3.5 Examples of IPC: POSIX Shared Memory

- POSIX Shared Memory
- Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm fd, 4096);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

IPC POSIX Producer/ Consumer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

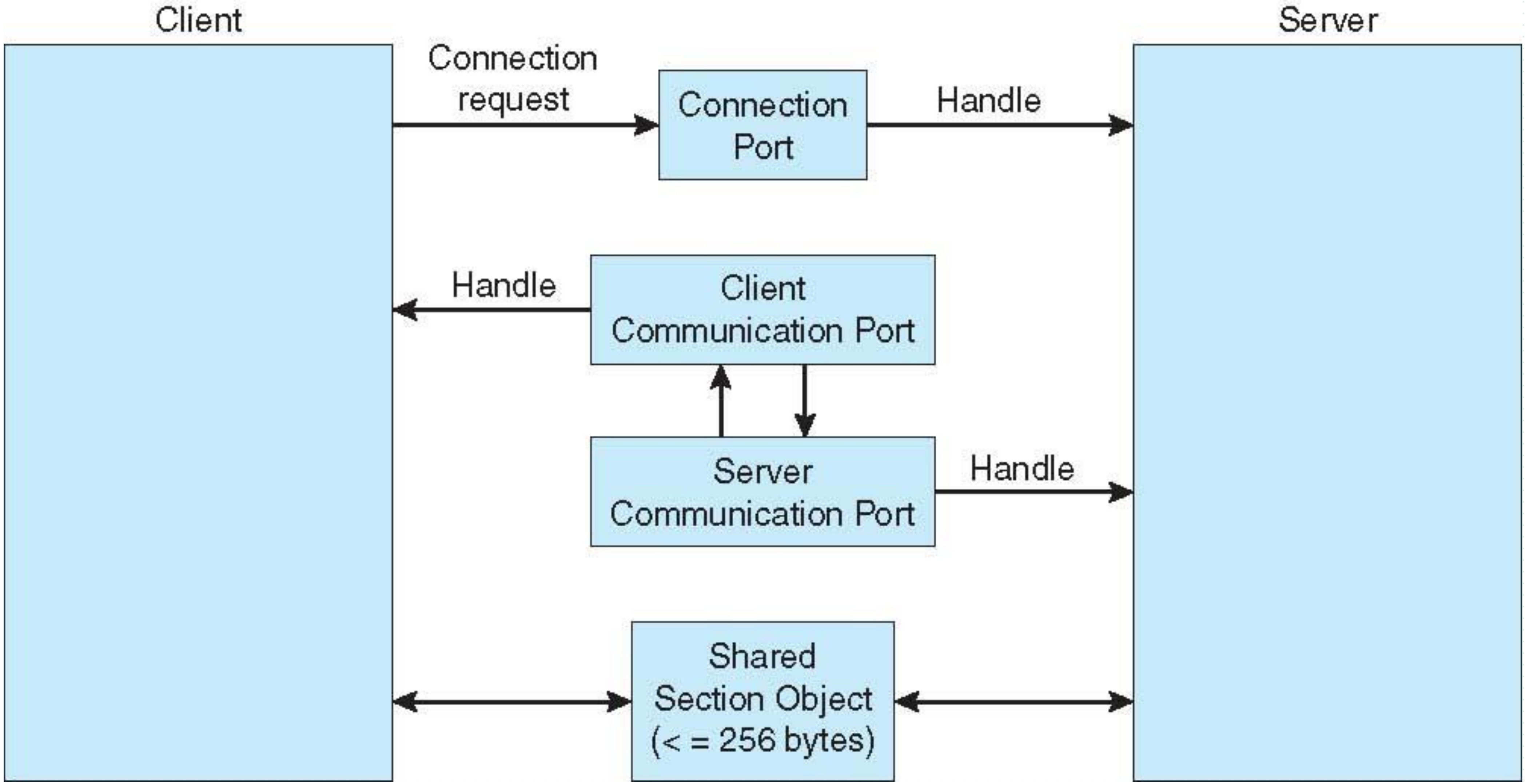
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

Local Procedure Calls in Windows

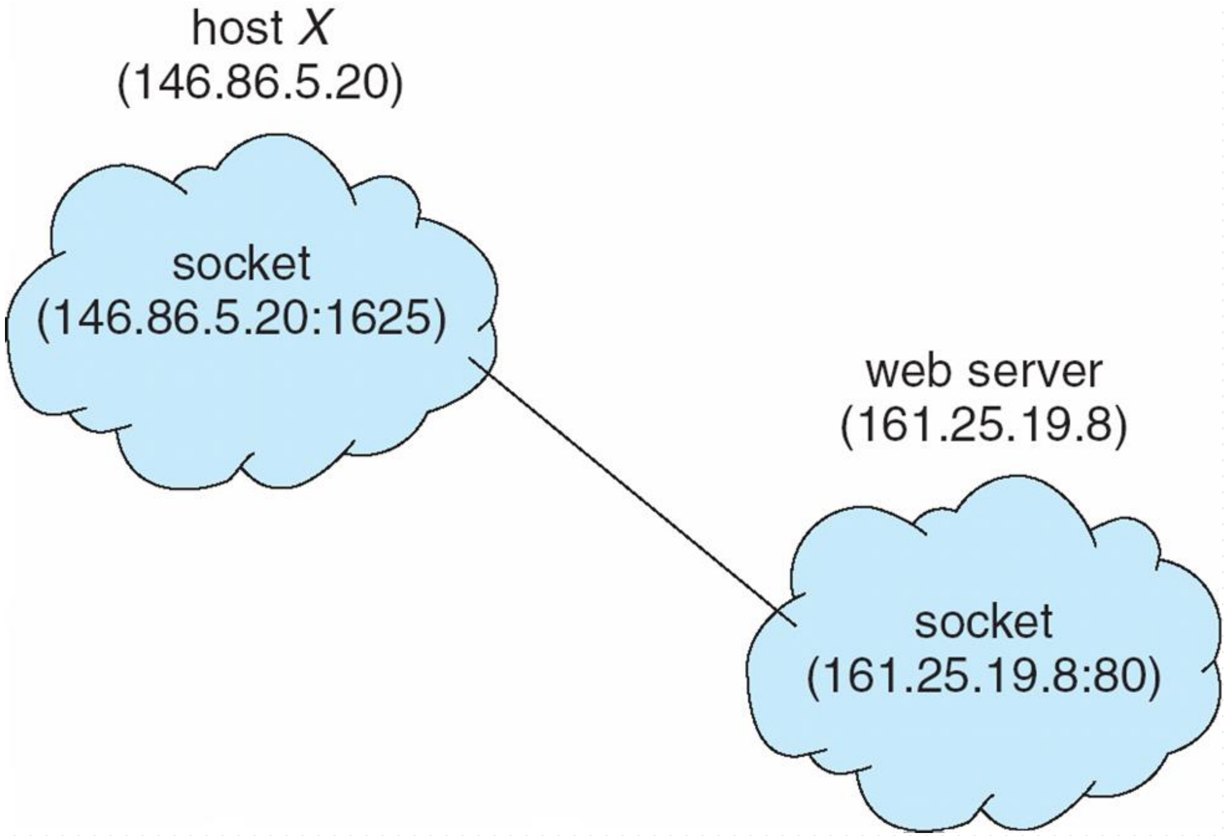


3.6 Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running



Sockets in Java

- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **`MulticastSocket`** class– data can be sent to multiple recipients

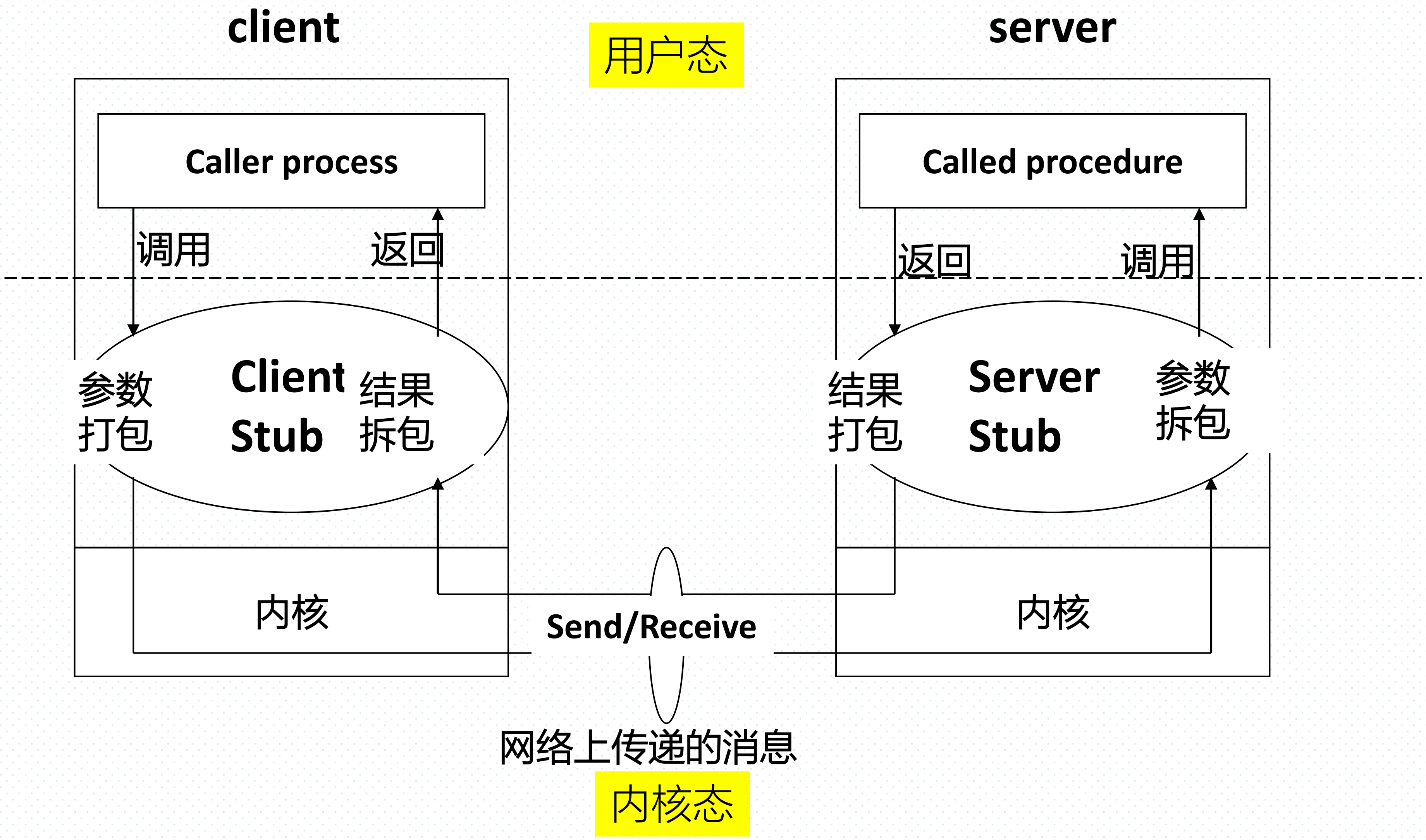
- Consider this "Date" server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls



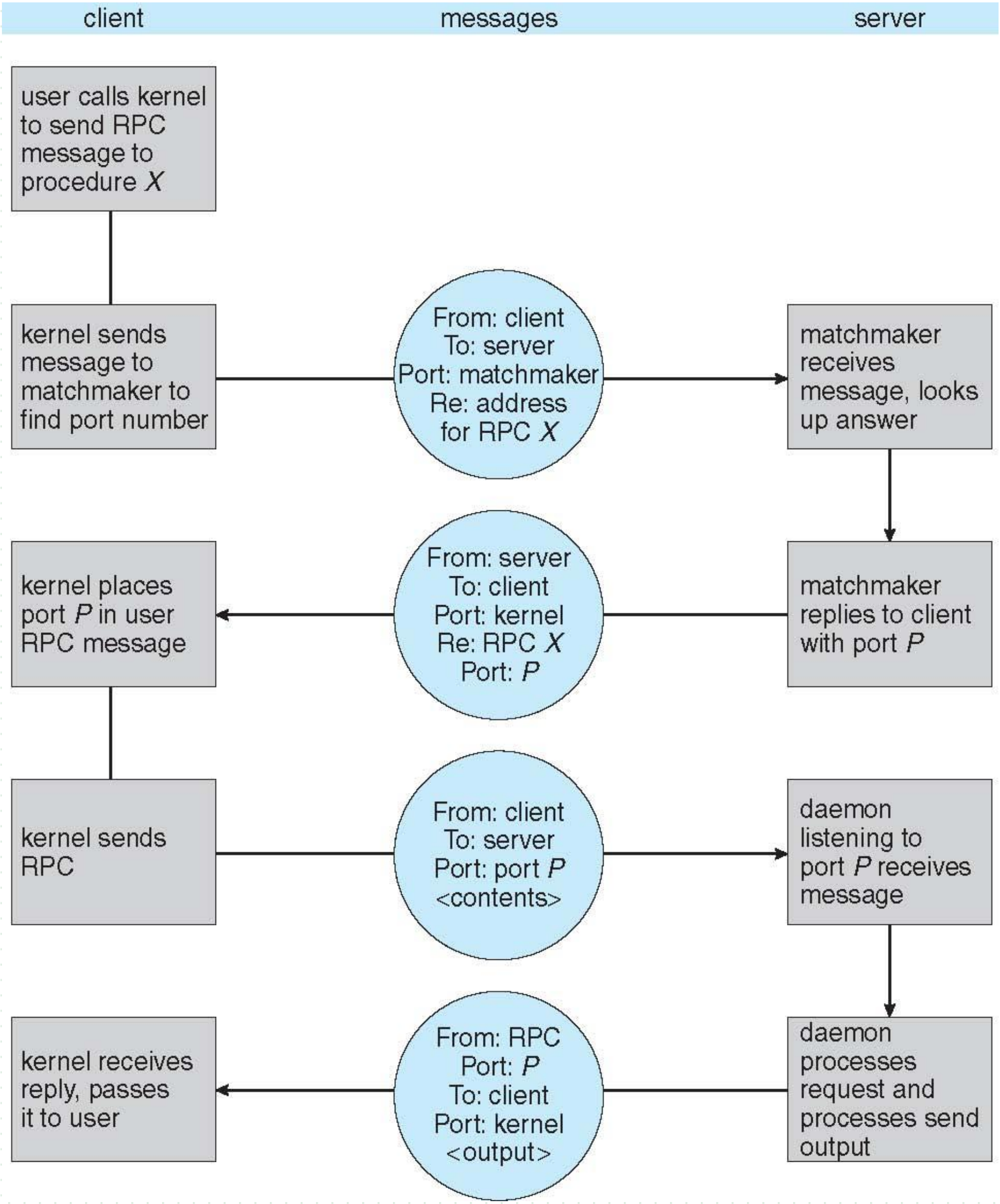
Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

RPC过程

- 客户程序（caller process）按通常的（类似于本地的）调用方式，调用客户存根
- 客户存根创建一个消息，封装参数，并陷入内核
- 内核将该消息发送给服务器端内核
- 服务器端内核将该消息传递给服务器存根
- 服务器存根从消息中获取参数，并调用服务器程序（called process）
- 服务器程序完成工作，将结果返回给服务器存根
- 服务器存根将结果打包进消息，并陷入OS内核。
- 服务器内核将消息返回给客户端内核
- 客户端内核将消息传递给客户存根
- 客户存根取出结果，返回给客户端调用程序

Remote Procedure Calls (Cont.)

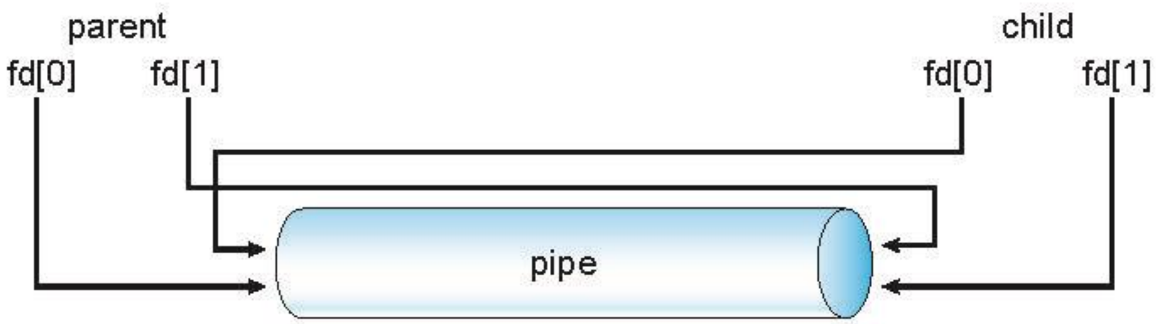


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



Windows calls these **anonymous pipes**

See Unix and Windows code samples in textbook

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

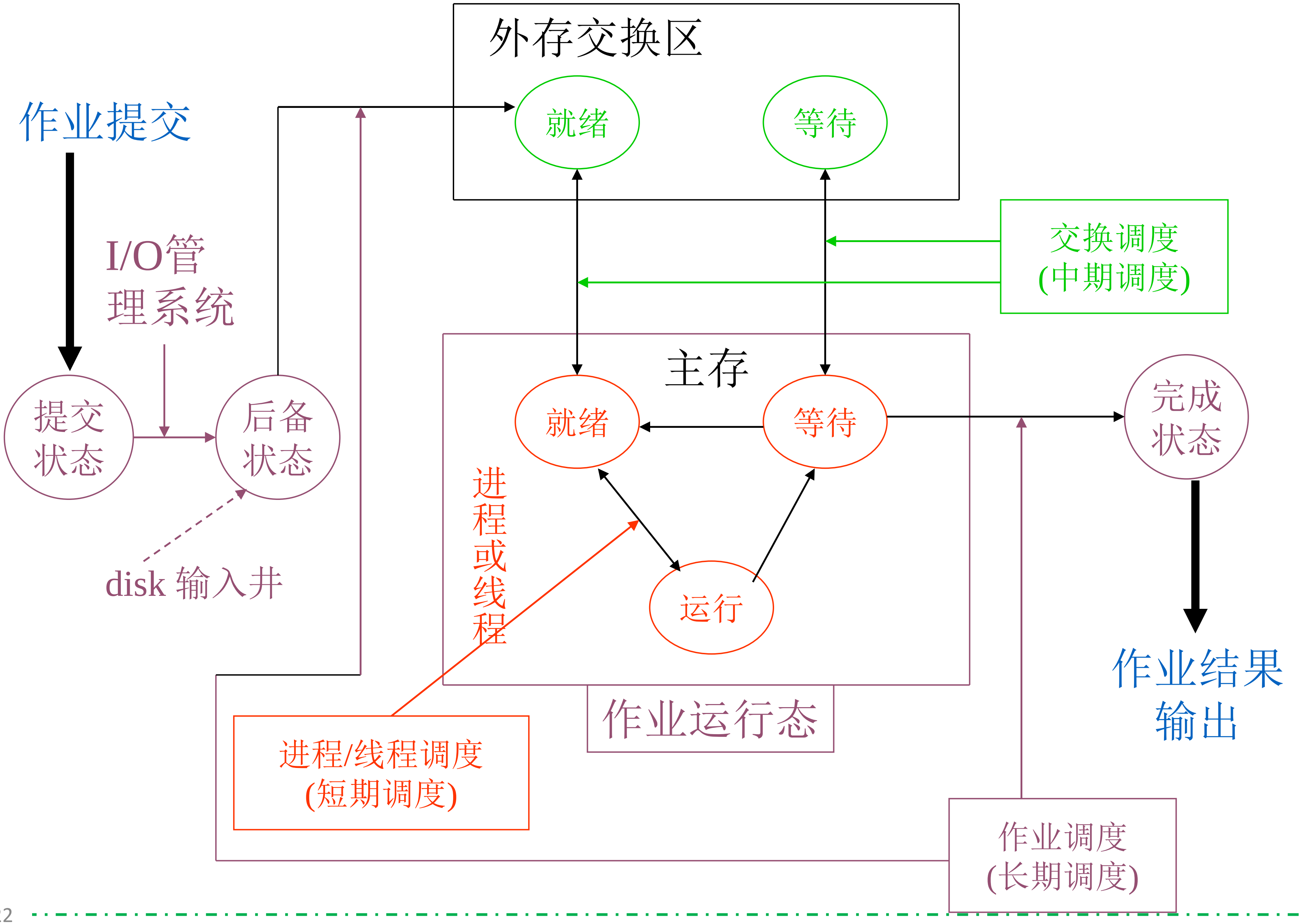
Appendix 3.A 调度的层次与作业调度

- 在大型通用系统和工作站中（多道批处理系统，分时系统），用户向系统提交作业，请求系统的服务
- 作业与进程的关系：
  - 作业是用户向计算机提交任务的任务实体
  - 进程/线程是OS为完成用户任务实体而设置的执行实体，是资源分配与处理器调度的基本单位
  - 一个作业可对应于多个执行进程/线程（根进程—子进程）

- 作业从进入系统到最终完成经过3个阶段
  - 作业通过I/O通道进入外部存储设备（磁盘，磁带机）上 的输入井
  - 长期调度程序调度作业进入主存，以进程/线程状态存在.
  - 进程/线程获得CPU，并运行，直至作业完成

调度的层次与作业调度

- 作业在整个生命周期过程中可划分为4个状态
  - 提交态：作业处于输入系统（提交）过程中
  - 后备/收容态：作业全部进入系统，处于外设输入井中等待运行
  - 运行态：作业被作业调度程序选中，进入主存，OS为其创建进程/线程并投入运行
  - 完成态：作业完成全部运行，释放所占用全部资源，准备退出系统。


外存交换区
就绪
等待
作业提交
I/O管理系统
交换调度
(中期调度)
主存
提交状态
后备状态
就绪
等待
完成状态
进程或线程
运行
disk 输入井
作业结果输出
作业运行态
进程/线程调度
(短期调度)
作业调度
(长期调度)

调度的层次与作业调度

- 作业调度
  - 又称为长期调度、宏观调度、高级调度。按照一定原则从输入井中的磁盘队列中选取作业进入主存,并分配必要资源。时间上通常是分钟、小时或天。
- 进程或线程调度
  - 又称为"微观调度"、"低级调度"。从CPU资源的角度，执行的基本单位的调度。时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。
- （内外存）交换调度
  - 又称为中期调度。将处于就绪态或等待态的某些进程在主存和外存交换区中倒换，以保证主存使用效率和进程执行效率。属于内存管理与扩充范畴

Thanks for your attention