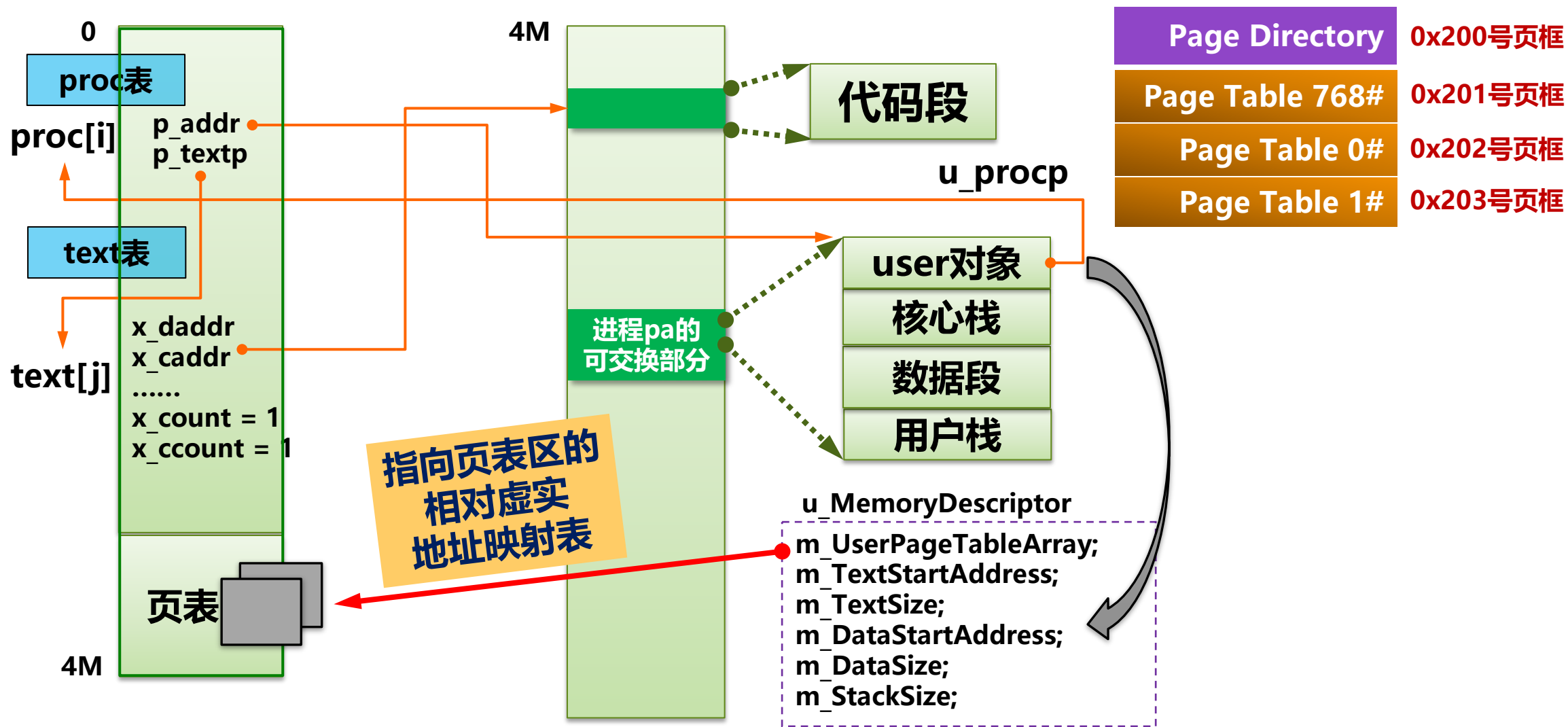


第三章

存储管理



现运行进程完整的进程图像





你必须知道的关于进程图像的细节

实例

UNIX V6++的地址变换

第一次执行.....创建进程pa

第二次执行.....创建进程pb

pa的相对虚地址映射表

Page Base Address	0x0	0x1	0x2	0x3	0x4
0x0	x	x	x	x	x
1024x	x	x	x	x	x
1025x	0	1	0	1	1025x
1026x	1	1	1	1	1026x
1027x	2	1	1	1	1027x
2047x	3	1	1	1	2047x

pb的相对虚地址映射表

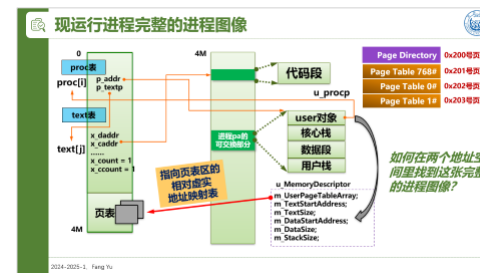
Page Base Address	0x0	0x1	0x2	0x3	0x4
0x0	x	x	x	x	x
1024x	x	x	x	x	x
1025x	0	1	0	1	1025x
1026x	1	1	1	1	1026x
1027x	2	1	1	1	1027x
2047x	3	1	1	1	2047x

两个进程有完全一样的程序地址(逻辑地址)空间

代码的共享



进程的切换



现运行进程的图像



UNIX V6++的地址变换



对代码的共享

```
#include <fcntl.h>
char buffer[2048];
int version = 1;
main( argc, argv)
int  argc;
char *argv[];
{
    int a, b;
    .....;
    sum(a, b);
    exit(0);
}
int sum( var1, var2)
int var1, var2;
{
    int count;
    count = var1 +var2;
    return(count);
}
```

第一次执行.....创建进程pa

第二次执行.....创建进程pb

pa的相对虚实地址映射表

	Page Base Address		s/u	r/w	p
0#	xxx		x	x	x

1024#	xxx		x	x	x
1025#	0		1	0	1
1026#	1		1	1	1
1027#	2		1	1	1
	全0				
2047#	3		1	1	1

pb的相对虚实地址映射表

	Page Base Address		s/u	r/w	p
0#	xxx		x	x	x

1024#	xxx		x	x	x
1025#	0		1	0	1
1026#	1		1	1	1
1027#	2		1	1	1
	全0				
2047#	3		1	1	1

两个进程有完全一样的程序地址（逻辑地址）空间



UNIX V6+ + 的地址变换



对代码的共享

p a 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x440		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x441		1	1	1
	0x442		1	1	1
1023#	0x443		1	1	1

p b 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x410		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x411		1	1	1
	0x412		1	1	1
1023#	0x413		1	1	1



UNIX V6+ + 的地址变换



对代码的共享

p
a
的
物
理
页
表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x440		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x441		1	1	1
	0x442		1	1	1
1023#	0x443		1	1	1

p
b
的
物
理
页
表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x410		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x411		1	1	1
	0x412		1	1	1
1023#	0x413		1	1	1

代码段物理页框号相同 → 两个进程共享代码段部分



UNIX V6+ + 的地址变换



对代码的共享

p a 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x440		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x441		1	1	1
	0x442		1	1	1
1023#	0x443		1	1	1

p b 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x410		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x411		1	1	1
	0x412		1	1	1
1023#	0x413		1	1	1

进程图像的可交换部分物理页框号不同 → 分别有自己的进程图像可交换部分



UNIX V6++的地址变换



main

...

L-2: push DWORD PTR [ebp-0x8]

L-1: push DWORD PTR [ebp-0x4]

L: call sum

L+1: add esp, 0x8

...

sum

T: push ebp

T+1: mov ebp, esp

T+2: sub esp, 0x10

完成加法计算

mov eax, DWORD PTR [ebp-0x4]

leave

ret

两个进程执行的是完全一样的代码



UNIX V6+ + 的地址变换



对代码的共享

p a 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x440		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x441		1	1	1
	0x442		1	1	1
1023#	0x443		1	1	1

p b 的物理页表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x410		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x411		1	1	1
	0x412		1	1	1
1023#	0x413		1	1	1

进程取指时:

若EIP内的地址为: $4M+7k$

$4M+7K \longrightarrow 4M+131K$: 代码段部分逻辑地址经地址变换后指向相同的物理地址



UNIX V6++的地址变换



对代码的共享

main

```
...  
L-2: push DWORD PTR [ebp-0x8]  
L-1: push DWORD PTR [ebp-0x4]
```

```
L:  call  sum
```

```
L+1: add  esp, 0x8  
...
```

sum

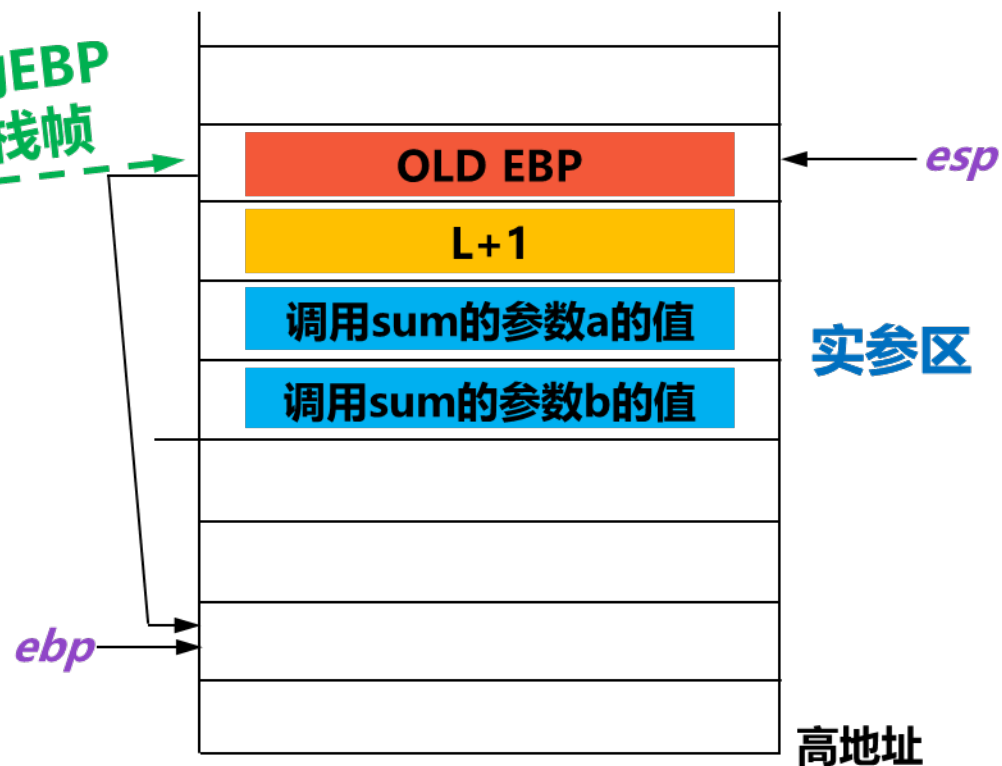
```
T:  push ebp  
T+1: mov  ebp, esp  
T+2: sub  esp, 0x10
```

完成加法计算

```
mov eax, DWORD PTR [ebp-0x4]  
leave  
ret
```

两个进程执行到这里时，
ESP和EBP中的值是相同的

前一栈帧的EBP
存入当前栈帧





UNIX V6++的地址变换



对代码的共享

p
a
的
物
理
页
表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x440		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x441		1	1	1
	0x442		1	1	1
1023#	0x443		1	1	1

ebp
esp

1023#

p
b
的
物
理
页
表

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	0x410		0	1	1

Page Table 1# (0x203号页框)

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	0x420		1	0	1
	0x411		1	1	1
	0x412		1	1	1
1023#	0x413		1	1	1

ebp
esp

1023#

两个进程有各自的进程图像可交换部分

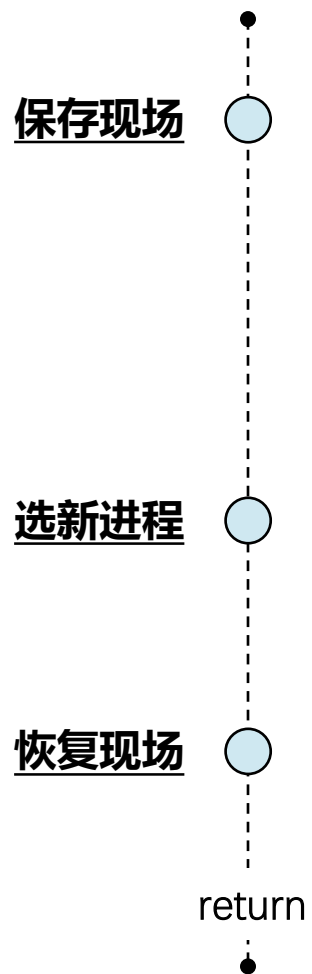
相同的逻辑地址经地址变换后指向不同的物理地址



现运行进程完整的进程图像



ProcessManager::Swch



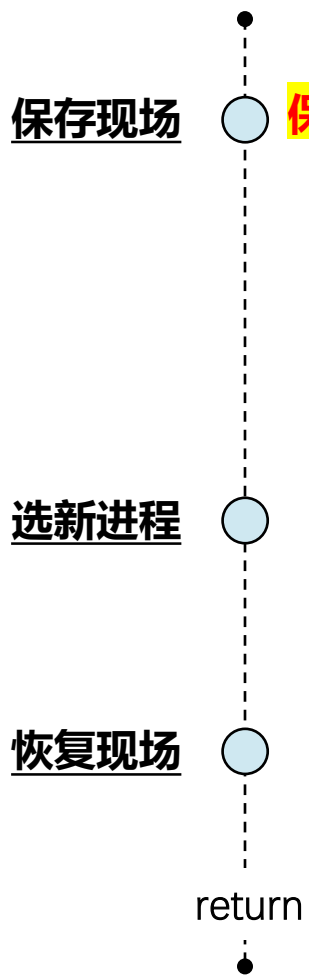


现运行进程完整的进程图像



进程上台建立页表

ProcessManager::Swch



保存什么?

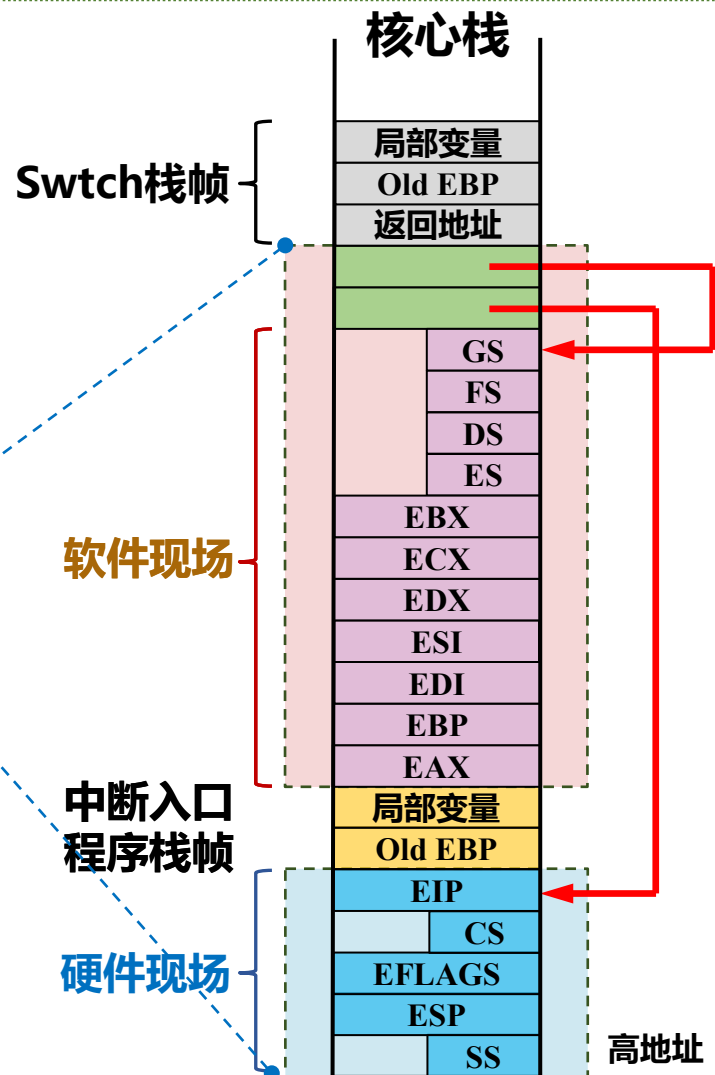
写入下台进程User结构中的
u_rsav[2]数组

esp
ebp

Swch的栈帧

一次中断响应的栈帧

进程下台时
保存在u_rsav[2]数组中的是逻辑地址



中断发生时现运行进程在用户态运行

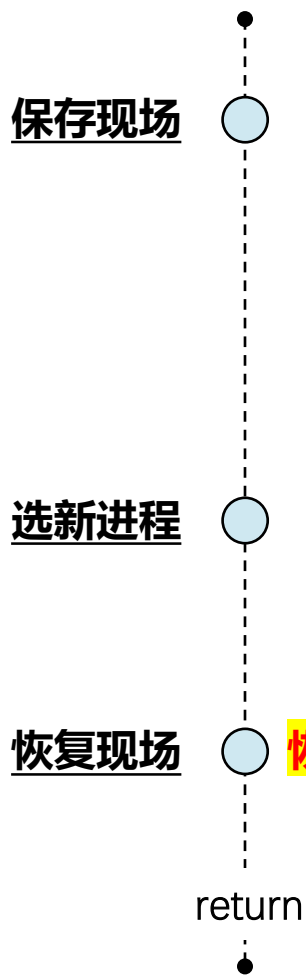


现运行进程完整的进程图像



进程上台建立页表

ProcessManager::Swch



恢复什么?

进程再上台时
恢复u_rsav[2]数组中的也是逻辑地址

将新进程 User 结构中的
u_rsav[2]数组中的值装入

esp
ebp

Swch的栈帧

一次中断响应的栈帧

指向新进程核心栈

Swch栈帧

软件现场

中断入口
程序栈帧

硬件现场

核心栈

局部变量

Old EBP

返回地址

GS

FS

DS

ES

EBX

ECX

EDX

ESI

EDI

EBP

EAX

局部变量

Old EBP

EIP

CS

EFLAGS

ESP

SS

高地址

中断发生时现运行进程在用户态运行

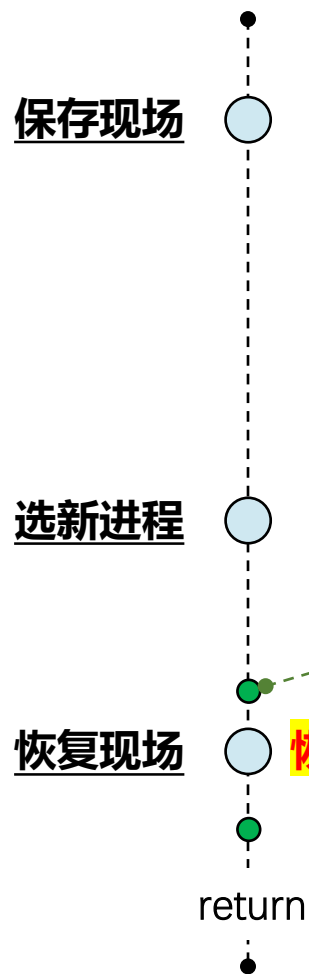


现运行进程完整的进程图像



进程上台建立页表

ProcessManager::Swch



这部分物理地址所有进程共享，进程切换时无需修改

为恢复现场准备好核心态地址变换

Page Directory (0x200号页框)

	Page Base Address		s/u	r/w	p
0#	0x202		1	1	1
1#	0x203		1	1	1
				
768#	0x201		0	1	1
				

Page Table 768# (0x201号页框)

	Page Base Address		s/u	r/w	p
0#	0		0	1	1
1#	1		0	1	1
				
1022#	1022		0	1	1
1023#	p_addr >> 12		0	1	1

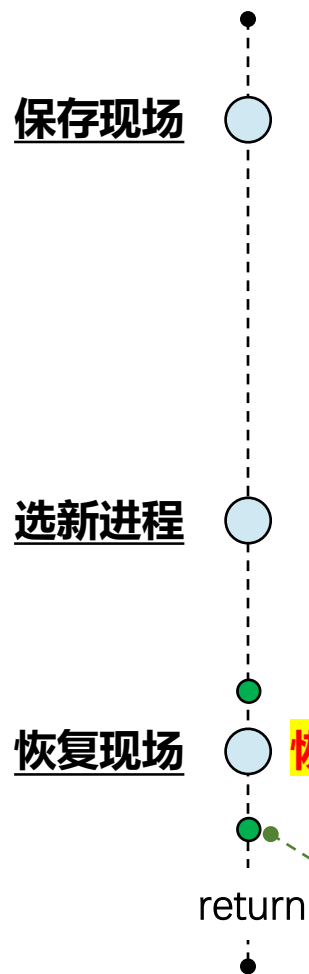


现运行进程完整的进程图像



进程上台建立页表

ProcessManager::Swch



相对虚实地址映射表

	Page Base Address	s/u	r/w	p
0#	xxx	x	x	x
...
1024#	xxx	if $r/w=0$, $x_caddr \gg 12$, 加上PBA		
n个页面的代码段	0	1	0	1
	1	1	0	1

	n - 1	1	0	1
m个页面的数据段	1	1	1	1

	m	1	1	1
	全0			
2047#	m + 1	1	1	1
	if $r/w=1$, $p_addr \gg 12$, 加上PBA			

Page Table 0# (0x202号页框)

Page Base Address	s/u	r/w	p
/	/	/	/

Page Table 1# (0x203号页框)

A

	Page Base Address		s/u	r/w	p
0#	/	/	/	/	/
1#	x_caddr>>12		1	0	1
	x_caddr>>12+1		1	0	1
	x_caddr>>12+(n-1)		1	0	1
	p_addr>>12+1		1	1	1
	p_addr>>12+m		1	1	1
	全0				
1023#	p_addr>>12+m+1		1	1	1

恢复什么?

为新进程建立用户态页表

为返回用户态准备好核心态地址变换



现运行进程完整的进程图像



进程上台建立页表

```
void MemoryDescriptor::MapToPageTable()
```

```
{
```

```
    User& u = Kernel::Instance().GetUser();
```

```
    PageTable* pUserPageTable = Machine::Instance().GetUserPageTableArray();
```

pUserPageTable指向两张用户态物理页表

```
    unsigned int textAddress = 0;
```

```
    if ( u.u_procp->p_textp != NULL )
```

```
    {
```

```
        textAddress = u.u_procp->p_textp->x_caddr;
```

```
    }
```

如果该进程有代码段，获取
代码段在内存的物理地址

```
    for (unsigned int i = 0; i < Machine::USER_PAGE_TABLE_CNT; i++)
```

```
    {
```

```
        for ( unsigned int j = 0; j < PageTable::ENTRY_CNT_PER_PAGETABLE; j++ )
```

```
        {
```

```
            pUserPageTable[i].m_Entrys[j].m_Present = 0; /* 先清0 */
```

```
            if ( 1 == this->m_UserPageTableArray[i].m_Entrys[j].m_Present )
```

如果在相对虚实地址映射表中的某一项P位不为0，则进入后续的页表处理

```
            {
```

```
                if ( 0 == this->m_UserPageTableArray[i].m_Entrys[j].m_ReadWriter )
```

```
                {
```

R/W位为0，对代码段的处理

```
                    pUserPageTable[i].m_Entrys[j].m_Present = 1;
```

```
                    pUserPageTable[i].m_Entrys[j].m_ReadWriter =
```

```
                        this->m_UserPageTableArray[i].m_Entrys[j].m_ReadWriter;
```

```
                    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress =
```

```
                        this->m_UserPageTableArray[i].m_Entrys[j].m_PageBaseAddress
```

```
                        + (textAddress >> 12);
```

```
                }
```

```
            }
```



现运行进程完整的进程图像



```
else if ( 1 == this->m_UserPageTableArray[i].m_Entrys[j].m_ReadWriter )
{
    pUserPageTable[i].m_Entrys[j].m_Present = 1;
    pUserPageTable[i].m_Entrys[j].m_ReadWriter =
        this->m_UserPageTableArray[i].m_Entrys[j].m_ReadWriter;
    pUserPageTable[i].m_Entrys[j].m_PageBaseAddress =
        this->m_UserPageTableArray[i].m_Entrys[j].m_PageBaseAddress
            + (u.u_procp->p_addr >> 12);
}
}
}
}
pUserPageTable[0].m_Entrys[0].m_Present = 1;
pUserPageTable[0].m_Entrys[0].m_ReadWriter = 1;
pUserPageTable[0].m_Entrys[0].m_PageBaseAddress = 0;

FlushPageDirectory();
}
```

R/W位为1, 对数据段的处理



现运行进程完整的进程图像



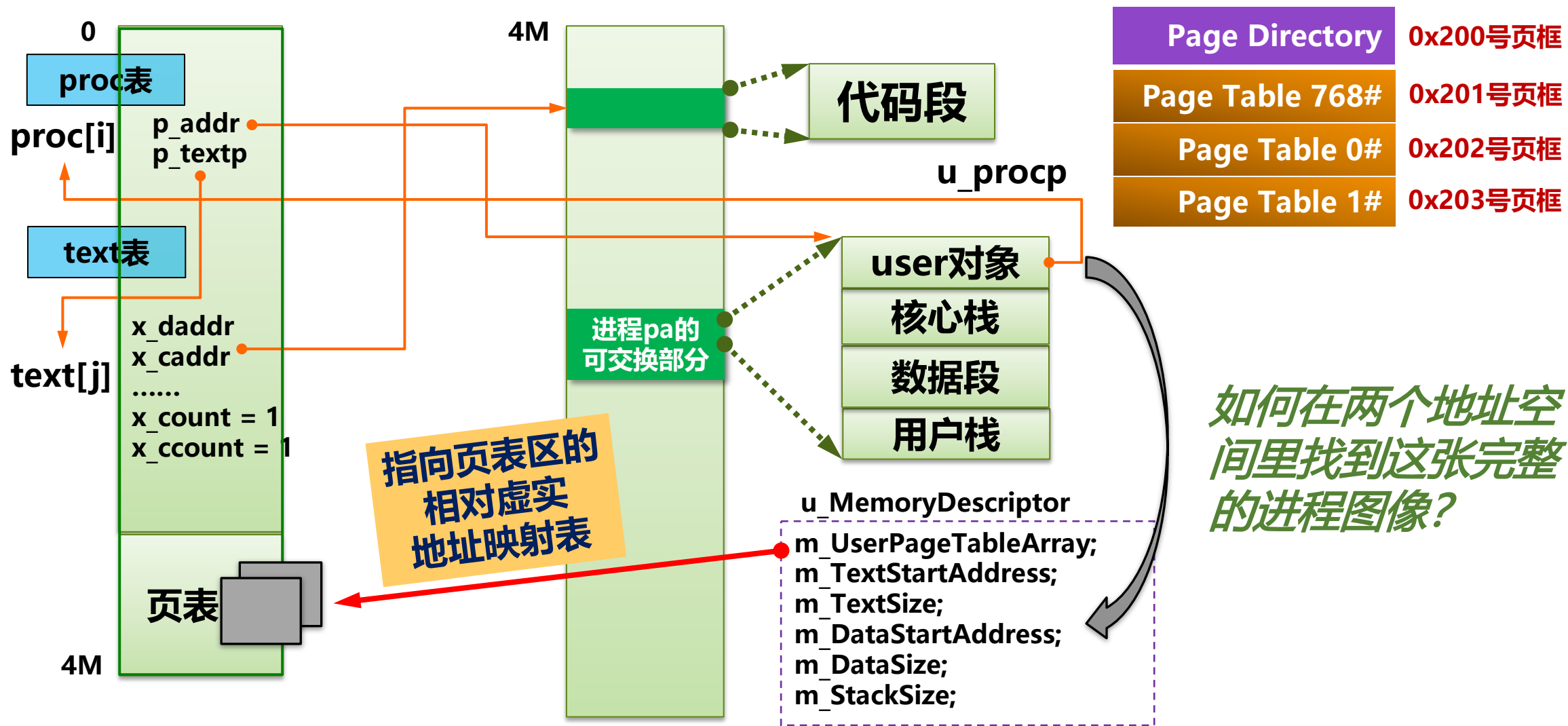
进程上台建立页表

ProcessManager::Swch





现运行进程完整的进程图像





现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段		
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC		
相对地址映射表		
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

从User结构的逻辑地址开始...



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从User结构的逻辑地址开始...

4M - 4K + 3G

```
static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;  
User& Kernel::GetUser()  
{  
    return *(User*) USER_ADDRESS; //获得现运行进程USER结构的逻辑地址  
}
```

```
if ( u.u_procp->p_textp != NULL )  
{  
    textAddress = u.u_procp->p_textp->x_caddr;  
}
```

```
User& u = Kernel::Instance().GetUser();  
u.u_ar0[User::EAX] = u.u_procp->p_pid;
```

```
u.u_ar0[User::EAX] = u.u_procp->p_pid;
```

```
u.u_procp->p_flag &= (~Process::STRC);
```

```
int dst = u.u_procp->p_addr + newSize - md.m_StackSize;
```



现运行进程完整的进程图像

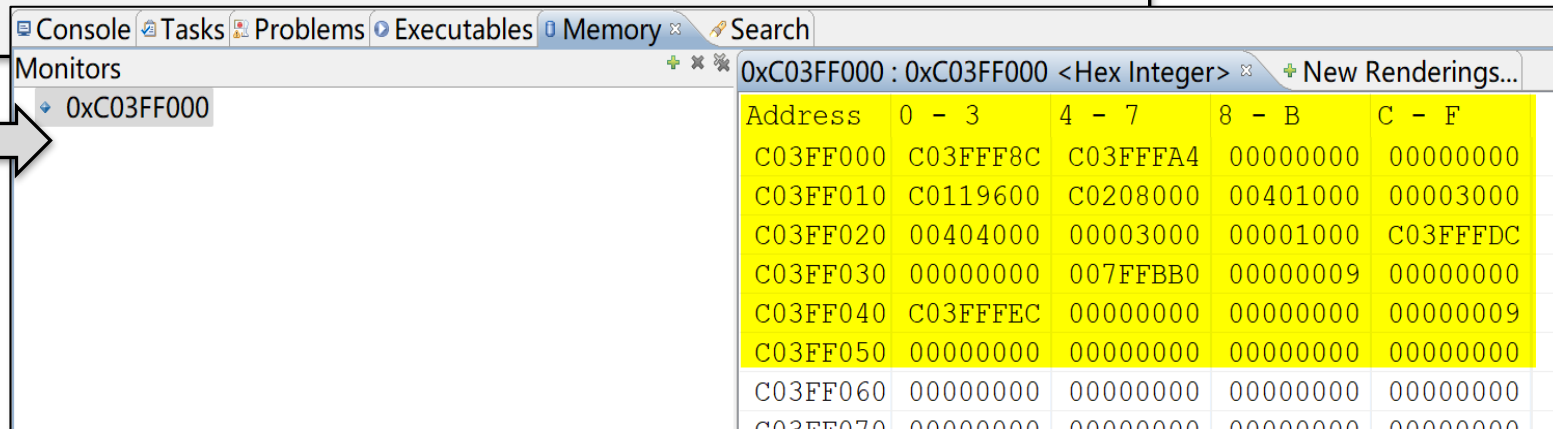


如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从User结构的逻辑地址开始...

4M - 4K + 3G

```
static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;  
User& Kernel::GetUser()  
{  
    return *(User*) USER_ADDRESS; //获得现运行进程USER结构的逻辑地址  
}
```



Address	0 - 3	4 - 7	8 - B	C - F
C03FF000	C03FFF8C	C03FFFA4	00000000	00000000
C03FF010	C0119600	C0208000	00401000	00003000
C03FF020	00404000	00003000	00001000	C03FFDC
C03FF030	00000000	007FFBB0	00000009	00000000
C03FF040	C03FFFE0	00000000	00000000	00000009
C03FF050	00000000	00000000	00000000	00000000
C03FF060	00000000	00000000	00000000	00000000
C03FF070	00000000	00000000	00000000	00000000



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从User结构的逻辑地址开始...

4M - 4K + 3G

```
static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;  
User& Kernel::GetUser()  
{  
    return *(User*) USER_ADDRESS; //获得现运行进程USER结构的逻辑地址  
}
```

Address	0 - 3	4 - 7	8 - B	C - F
C03FF000	C03FFF8C	C03FFFA4	00000000	00000000
C03FF010	C0119600	C0208000	00401000	00003000
C03FF020	00404000	00003000	00001000	C03FFFD0
C03FF030	00000000	007FFBB0	00000009	00000000
C03FF040	C03FFFE0	00000000	00000000	00000009
C03FF050	00000000	00000000	00000000	00000000
C03FF060	00000000	00000000	00000000	00000000
C03FF070	00000000	00000000	00000000	00000000

```
class User  
{  
    .....;  
public:  
    unsigned long u_rsav[2]; /* 用于保存esp与ebp指针 */  
    unsigned long u_ssav[2]; /* 用于对esp和ebp指针的二次保护 */  
    Process* u_procp; /* 指向该u结构对应的Process结构 */  
    MemoryDescriptor u_MemoryDescriptor;  
    .....;  
};
```




现运行进程完整的进程图像



跟踪现运行进程图像

如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从User结构的逻辑地址开始...

4M - 4K + 3G

```
static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;  
User& Kernel::GetUser()  
{  
    return *(User*) USER_ADDRESS; //获得现运行进程USER结构的逻辑地址  
}
```

Address	0 - 3	4 - 7	8 - B	C - F
C03FF000	C03FFF8C	C03FFFA4	00000000	00000000
C03FF010	C0119600	C0208000	00401000	00003000
C03FF020	00404000	00003000	00001000	C03FFFD0
C03FF030	00000000	007FFBB0	00000009	00000000
C03FF040	C03FFFE0	00000000	00000000	00000009
C03FF050	00000000	00000000	00000000	00000000
C03FF060	00000000	00000000	00000000	00000000
C03FF070	00000000	00000000	00000000	00000000

```
class User  
{  
    .....;  
public:  
    unsigned long u_rsav[2]; /* 用于保存esp与ebp指针 */  
    unsigned long u_ssav[2]; /* 用于对esp和ebp指针的二次保护 */  
    Process* u_procp; /* 指向该u结构对应的Process结构 */  
    MemoryDescriptor u_MemoryDescriptor;  
    .....;  
};
```

确定u_procp的值 (proc结构的逻辑地址)



现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段		
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	
相对地址映射表		
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

0xC0119600



现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段		
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	u_procp-3G
相对地址映射表		
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

0xC0119600

0x00119600



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从User结构的逻辑地址开始...

4M - 4K + 3G

```
static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;  
User& Kernel::GetUser()  
{  
    return *(User*) USER_ADDRESS; //获得现运行进程USER结构的逻辑地址  
}
```

Monitors

0xC03FF000

0xC03FF000 : 0xC03FF000 <Hex Integer> + New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
C03FF000	C03FFF8C	C03FFFA4	00000000	00000000
C03FF010	C0119600	C0208000	00401000	00003000
C03FF020	00404000	00003000	00001000	C03FFFD0
C03FF030	00000000	007FFBB0	00000009	00000000

```
class User  
{  
    .....;  
public:  
    unsigned long u_rsav[2]; /* 用于保存esp与ebp指针 */  
    unsigned long u_ssav[2]; /* 用于对esp和ebp指针的二次保护 */  
    Process* u_procp; /* 指向该u结构对应的Process结构 */  
    MemoryDescriptor u_MemoryDescriptor; .....;  
};
```

```
class MemoryDescriptor  
{  
    .....;  
public:  
    PageTable* m_UserPageTableArray;  
    unsigned long m_TextStartAddress; /* 代码段起始地址 */  
    unsigned long m_TextSize; /* 代码段长度 */  
    unsigned long m_DataStartAddress; /* 数据段起始地址 */  
    unsigned long m_DataSize; /* 数据段长度 */  
    unsigned long m_StackSize; /* 栈段长度 */  
};
```

跟踪现运行进程图像



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段		
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	u_procp-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

0xC0208000 = 3G + 2M + 32K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段		
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	u_procp-3G
		0xC0208000 = 3G + 2M + 32K
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
		2M + 32K
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段	<code>u_MemoryDescriptor.m_DataStartAddress</code>	
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	<code>u_procp</code>	<code>u_procp-3G</code>
相对地址映射表	<code>u_MemoryDescriptor.m_UserPageTableArray</code>	<code>u_MemoryDescriptor.m_UserPageTableArray-3G</code>
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

0x00404000 = 4M + 16K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段	u_MemoryDescriptor.m_DataStartAddress	
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	u_procp-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

跟踪现运行进程图像



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从Proc结构的逻辑地址开始...

Address	0 - 3	4 - 7	8 - B	C - F
C0119600	00000000	00000002	00000001	0040F000
C0119610	00005000	C011AE94	00000003	00000001
C0119620	00000065	00000019	00000000	00000000
C0119630	00000000	00000000	C0120DA0	00000000
C0119640	00000000	00000000	FFFFFFFF	00000000
C0119650	00000000	00000000	00000000	00000000
C0119660	00000000	00000000	00000000	00000000
C0119670	00000000	00000000	00000000	00000000
C0119680	00000000	00000000	FFFFFFFF	00000000
C0119690	00000000	00000000	00000000	00000000

```
class Process
{
    .....;
public:
    short p_uid;
    int p_pid;
    int p_ppid;

    unsigned long p_addr;
    unsigned int p_size;
    Text*p_textp;

    ProcessState p_stat;
    int p_flag;
    int p_pri;
    int p_cpu;
    int p_nice;
    int p_time;
    unsigned long p_wchan;
    .....;
};
```

现运行进程为2#进程，父进程为1#进程



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从Proc结构的逻辑地址开始...

Monitors				
0xc0119600 : 0xC0119600 <Hex Integer> x New Renderings...				
0xc03ff000				
0xc0119600				
0x00404000				
Address	0 - 3	4 - 7	8 - B	C - F
C0119600	00000000	00000002	00000001	0040F000
C0119610	00005000	C011AE94	00000003	00000001
C0119620	00000065	00000019	00000000	00000000
C0119630	00000000	00000000	C0120DA0	00000000
C0119640	00000000	00000000	FFFFFFFF	00000000
C0119650	00000000	00000000	00000000	00000000
C0119660	00000000	00000000	00000000	00000000
C0119670	00000000	00000000	00000000	00000000
C0119680	00000000	00000000	FFFFFFFF	00000000
C0119690	00000000	00000000	00000000	00000000
C01196A0	00000000	00000000	00000000	00000000

```
class Process
{
    .....;
public:
    short p_uid;
    int p_pid;
    int p_ppid;

    unsigned long p_addr;
    unsigned int p_size;
    Text*p_textp;

    ProcessState p_stat;
    int p_flag;
    int p_pri;
    int p_cpu;
    int p_nice;
    int p_time;
    unsigned long p_wchan;
    .....;
};
```

现运行进程为2#进程，父进程为1#进程

跟踪现运行进程图像



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段	u_MemoryDescriptor.m_DataStartAddress	
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	u_procp->p_addr 0x0040F000
PROC	u_procp	u_procp-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	0x00410000
数据段	u_MemoryDescriptor.m_DataStartAddress	u_procp->p_addr+4K
堆栈段	8M-4K	
		0x0040F000
PPDA (USER)	3G+4M-4K	u_procp->p_addr
PROC	u_procp	u_procp-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

跟踪现运行进程图像

	逻辑地址空间	物理地址空间
代码段	4M+4K	0x00410000
数据段	u_MemoryDescriptor.m_DataStartAddress	u_procp->p_addr+4K
堆栈段	8M-4K	p_addr+4K+u_MemoryDescriptor.m_DataSize
		0x00413000
PPDA (USER)	3G+4M-4K	u_procp->p_addr
PROC	u_procp	u_procp-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

跟踪现运行进程图像

	逻辑地址空间	物理地址空间
代码段	4M+4K	0x00410000
数据段	<code>u_MemoryDescriptor.m_DataStartAddress</code>	<code>u_procp->p_addr+4K</code> 0x00413000
堆栈段	8M-4K	<code>p_addr+4K+u_MemoryDescriptor.m_DataSize</code>
PPDA (USER)	3G+4M-4K	0x0040F000 <code>u_procp->p_addr</code>
PROC	<code>u_procp</code>	<code>u_procp-3G</code>
相对地址映射表	<code>u_MemoryDescriptor.m_UserPageTableArray</code>	<code>u_MemoryDescriptor.m_UserPageTableArray-3G</code>
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



如何跟踪现运行进程在两个地址空间的用户态下完整的进程图像？

从Proc结构的逻辑地址开始...

Address	0 - 3	4 - 7	8 - B	C - F
C0119600	00000000	00000002	00000001	0040F000
C0119610	00005000	C011AE94	00000003	00000001
C0119620	00000065	00000019	00000000	00000000
C0119630	00000000	00000000	C0120DA0	00000000
C0119640	00000000	00000000	FFFFFFFF	00000000
C0119650	00000000	00000000	00000000	00000000
C0119660	00000000	00000000	00000000	00000000
C0119670	00000000	00000000	00000000	00000000
C0119680	00000000	00000000	FFFFFFFF	00000000
C0119690	00000000	00000000	00000000	00000000

```
class Process
{
    .....;
public:
    short p_uid;
    int p_pid;
    int p_ppid;

    unsigned long p_addr;
    unsigned int p_size;
    Text*p_textp;

    ProcessState p_stat;
    int p_flag;
    int p_pri;
    int p_cpu;
    int p_nice;
    int p_time;
    unsigned long p_wchan;
    .....;
};
```

跟踪现运行进程图像



现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

	逻辑地址空间	物理地址空间
代码段	4M+4K	<code>u_procp->p_textp->x_caddr</code>
数据段	<code>u_MemoryDescriptor.m_DataStartAddress</code>	<code>u_procp->p_addr+4K</code>
堆栈段	8M-4K	<code>p_addr+4K+u_MemoryDescriptor.m_DataSize</code>
PPDA (USER)	3G+4M-4K	<code>u_procp->p_addr</code>
PROC	<code>u_procp</code>	<code>u_procp-3G</code>
相对地址映射表	<code>u_MemoryDescriptor.m_UserPageTableArray</code>	<code>u_MemoryDescriptor.m_UserPageTableArray-3G</code>
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K



现运行进程完整的进程图像



在进程的整个生命周期中，下面哪些地址的值会发生变化？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段	u_MemoryDescriptor.m_DataStartAddress	
堆栈段	8M-4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	
物理页表	3G+2M ~ 3G+2M+16K	



现运行进程完整的进程图像



在进程的整个生命周期中，下面哪些地址的值会发生变化？

	逻辑地址空间	物理地址空间
代码段	4M+4K	
数据段	u_MemoryDescriptor.m_DataStartAddress	
堆栈段	8M-4K	
	如果程序执行过程中，堆栈深度不够，则追加4K	
PPDA (USER)	3G+4M-4K	
PROC	u_procp	
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	
物理页表	3G+2M ~ 3G+2M+16K	



现运行进程完整的进程图像



在进程的整个生命周期中，下面哪些地址的值会发生变化？

	逻辑地址空间	物理地址空间
代码段	4M+4K	u_procp->p_textp->x_caddr
数据段	u_MemoryDescriptor.m_DataStartAddress	u_procp->p_addr+4K
堆栈段	8M-4K	p_addr+4K+u_MemoryDescriptor.m_DataSize
PPDA (USER)	3G+4M-4K	u_procp->p_addr
PROC	u_procp	p-3G
相对地址映射表	u_MemoryDescriptor.m_UserPageTableArray	u_MemoryDescriptor.m_UserPageTableArray-3G
物理页表	3G+2M ~ 3G+2M+16K	2M ~ 2M+16K

1. 进程图像每次换进换出
2. 堆栈段变化



现运行进程完整的进程图像



如何跟踪**现运行进程**在两个地址空间的用户态下完整的进程图像？

```
User& u = Kernel::Instance().GetUser();
```

```
u.u_ar0[User::EAX] = u.u_procp->p_pid;
```

```
if ( u.u_procp->p_textp != NULL )  
{  
    textAddress = u.u_procp->p_textp->x_caddr;  
}
```

```
u.u_procp->p_flag &= (~Process::STRC);
```

```
int dst = u.u_procp->p_addr + newSize - md.m_StackSize;
```

主要内容

3.1 存储管理的主要任务

3.2 连续分配方式

3.3 页式存储管理

3.4 段式与段页式存储管理**

3.5 **UNIX 存储管理**

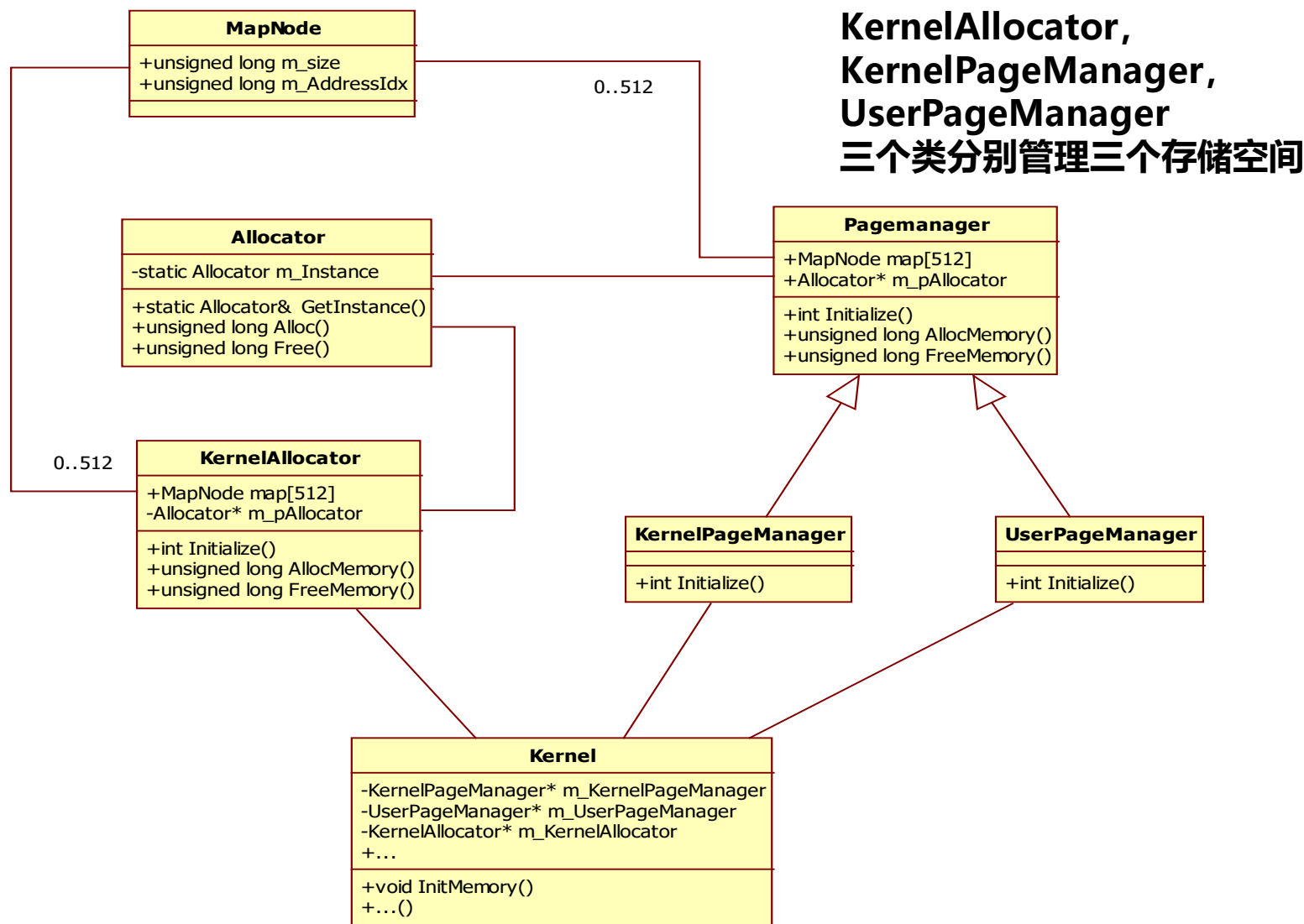
- 程序地址空间
- 物理地址空间
- 地址变换
- **存储空间管理**



存储空间管理



与存储管理相关的类

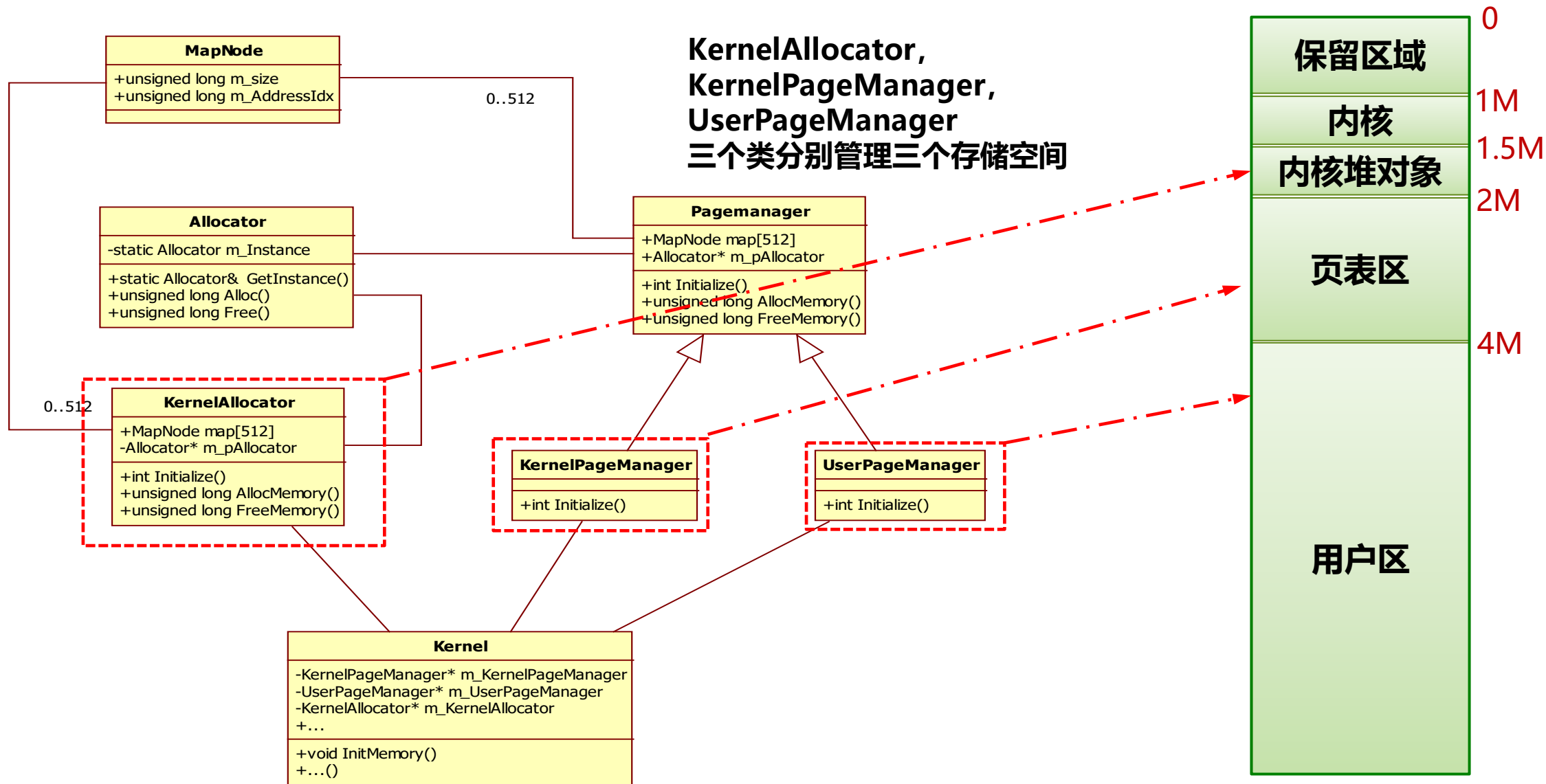




存储空间管理



与
存储
管理
相关
的
类

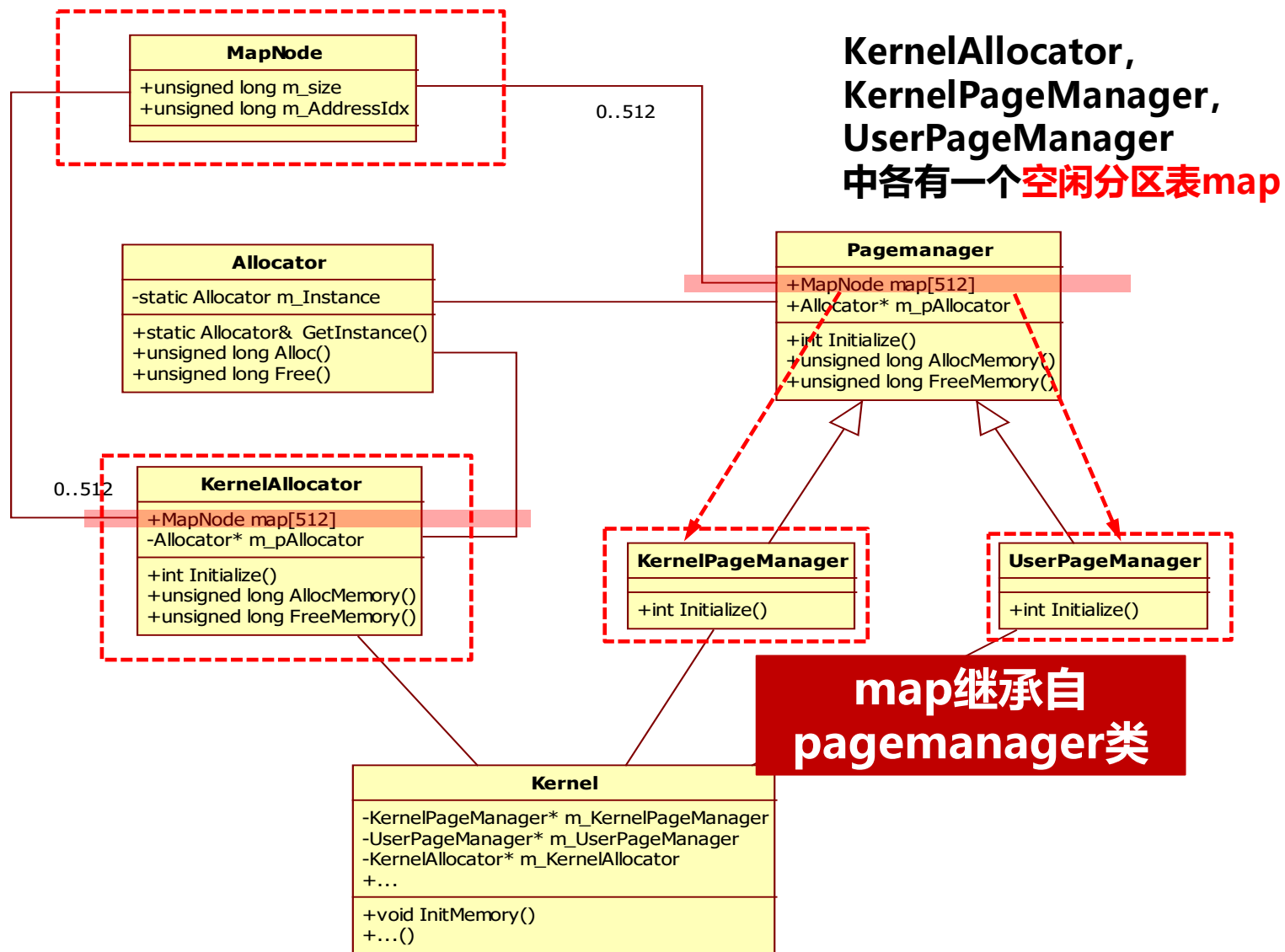




存储空间管理



与
存储
管理
相关
的
类





存储空间管理



内核对象区、页表区、用户区均采用**可变分区方式**进行分配，
分别由**三张空闲分区表**管理：

`public:MapNode map[512];` 最多包含512个表项的空闲分区表

`struct MapNode`

```
{  
    unsigned long m_Size;           其中每一个表项包含空闲分  
    unsigned long m_AddressIdx; 区的大小和分区的起始地址  
};
```

内核堆对象区，其map初始化为：

`map[0]. m_AddressIdx = 1.5M, map[0]. m_Size = 0.5M。`

负责管理页表区，其map初始化为：

`map[0]. m_AddressIdx = 2M+16K`，即：从204号页框开始。

`map[0]. m_Size = 2M-16K`，即：剩余的页表区。

用户区，其map初始化为：

`map[0]. m_AddressIdx = 4M`，大小至整个物理内存。

大小	起始地址
0.5M	1.5M

大小	起始地址
2M-16K	2M+16K

大小	起始地址
至整个物理内存	4M

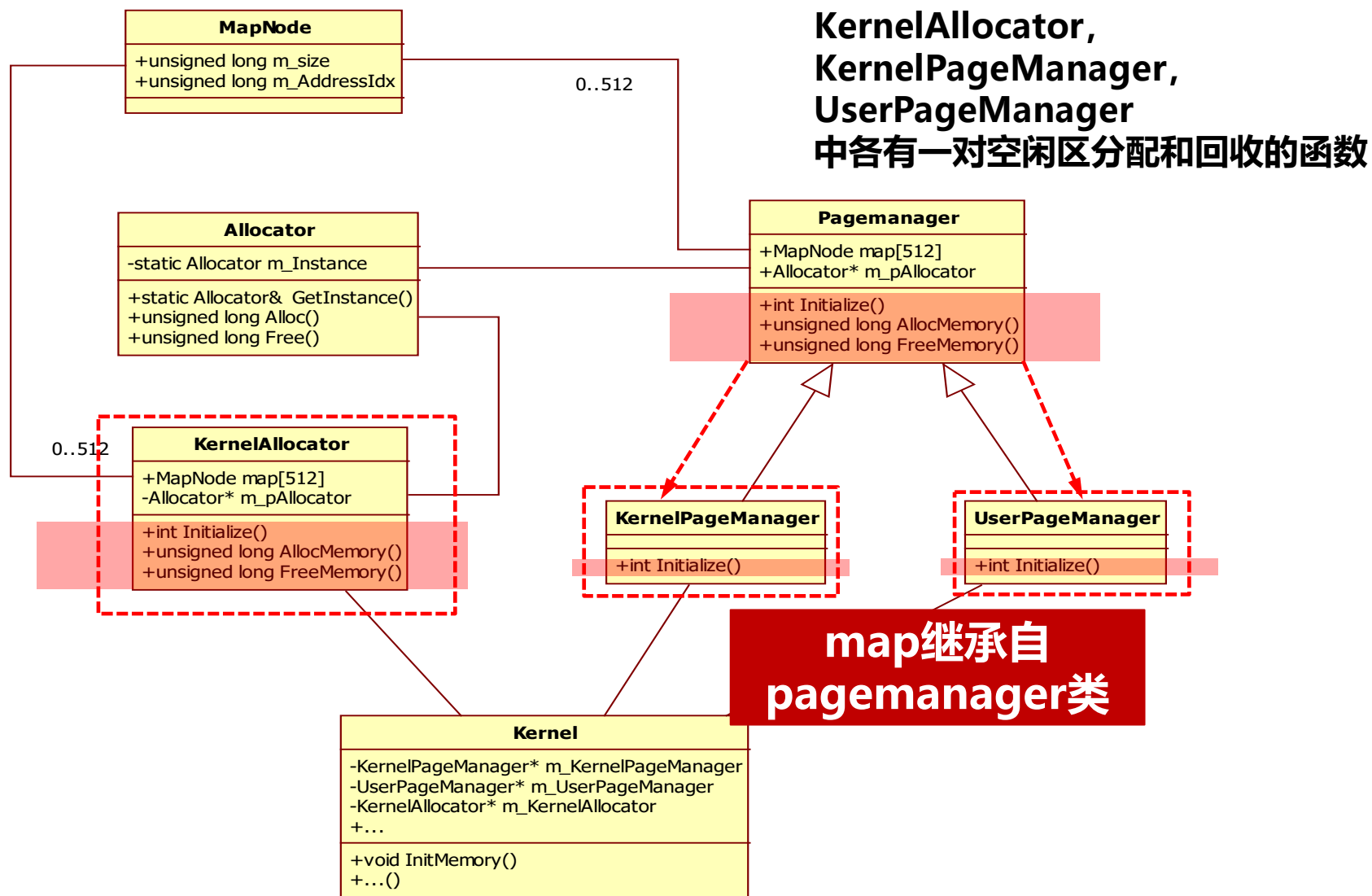




存储空间管理



与存储管理相关的类





存储空间管理



内核对象区、页表区、用户区中各有一对空闲区分配和回收的函数

回收一个大小为size, 起始地址为addIdx的分区到map中

按首次适应算法在map中分配一个大小为size的分区

```
public:
unsigned long AllocMemory(unsigned long size);
unsigned long FreeMemory(unsigned long size, unsigned long addIdx);
```

分配时

KernelAllocator:
size = 实际需求

按首次适应
算法分配

KernelPageManager:
size = 两个页框 (8K)

UserPageManager:
size = 向上取整 (实际需求/4k) × 4k
即: 满足需求的4K整数倍。

回收时

KernelAllocator:
size = 实际占用

考虑四种回收分
区的位置情况

KernelPageManager:
size = 两个页框 (8K)

UserPageManager:
size = 实际占用

保留区域

内核

内核堆对象

页表区

用户区

0

1M

1.5M

2M

4M



存储空间管理



进程创建时:

和父进程共享已经在内存的代码段:

实际需求 = 可交换部分

代码段与可交换部分不连续

在页表区中申请**两个连续的空闲页框**，建立相对虚实地址映射表

1

`KernelPageManager::Alloc(map[], 8K);`

按进程**实际需求**在用户区中申请连续内存

2

`UserPageManager::Alloc(map[], 向上取整 (实际需求/4k) × 4k);`





存储空间管理



进程图像交换时:

最后一个使用该代码段的进程:
一起释放代码段
否: 不释放

第一个使用该代码段的进程:
实际需求 = 代码段 + 可交换部分

代码段与可交换部分连续

共享一个已经在内存的代码段:

实际需求 = 可交换部分

代码段与可交换部分不连续

换出进程释放**曾经占用的内存**

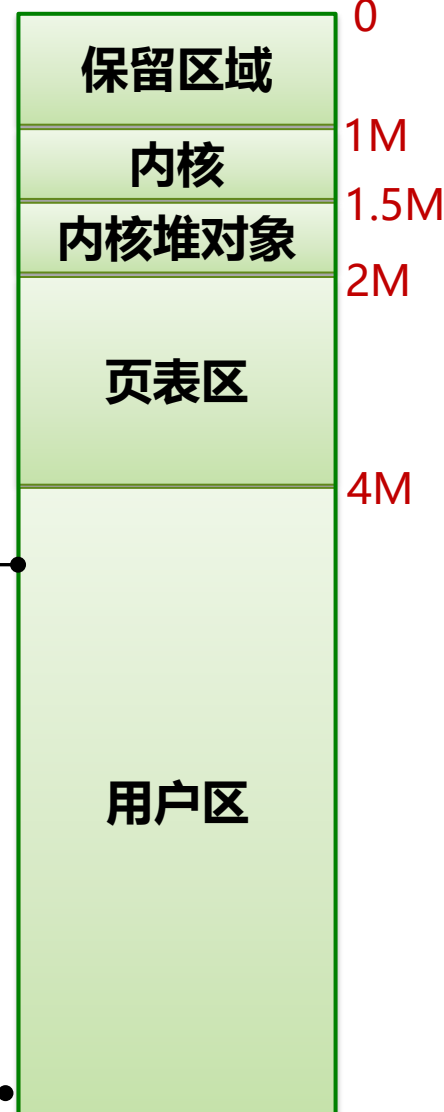
1

```
UserPageManager::Free(map[], p_size, p_addr);  
UserPageManager::Free(map[], x_size, x_caddr);
```

为换入进程**实际需求**在用户区中申请内存

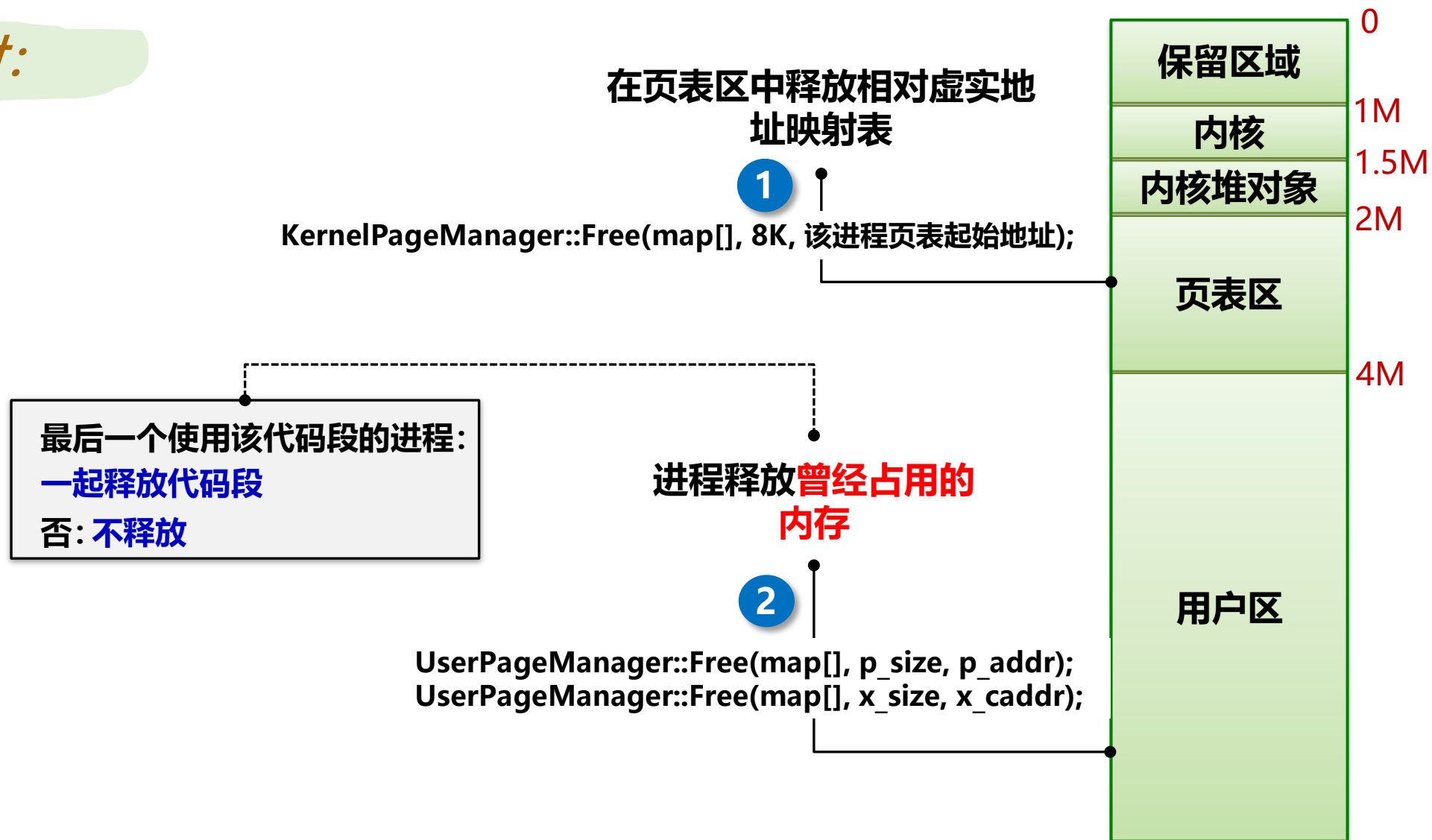
2

```
UserPageManager::Alloc(map[], 向上取整 (实际需求/4k) × 4k );
```





进程终止时:





本节小结



1 UNIX V6++对存储空间的管理

2 UNIX V6++进程生命周期中和内存管理相关的操作

阅读教材：154页 ~ 164页



E09：存储管理（UNIX存储管理）



P03：UNIX V6++完整的进程图象