OPERATING SYSTEMS

Introduction to Process

Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

Introduction to Process

- **目录**
  - 基本概念
  - 进程表和进程控制块
  - 进程状态和转换
  - 进程操作
    - 进程创建
    - 进程终止
  - Unix和Linux示例
  - 进程调度
  - 进程切换

Basic Concepts

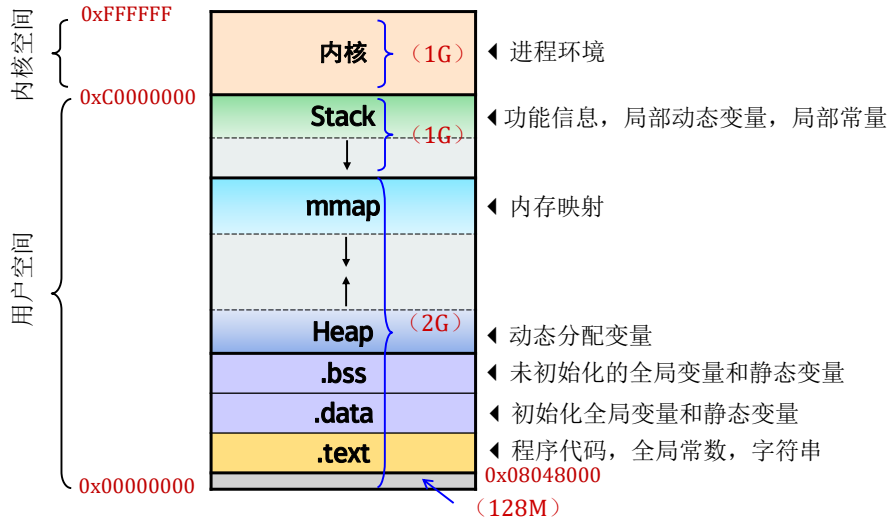
- **基本概念**
  - 进程是一个正在执行的程序；构成所有计算的基础。进程执行必须按顺序进行。
    - 程序是一个被动实体，包含作为可执行文件存储在磁盘上的指令列表。
    - 进程是一个活动实体，具有相应程序的某些规范和一组相关资源。
    - 当一个程序的可执行对象被加载到内存中并交给进程调度器时，该程序就变成了一个进程。
    - 进程是正在运行的程序的实例；它可以分配给处理器并在处理器上执行。
  - 可以通过CLI输入程序名称、GUI鼠标单击等来启动程序的执行。
  - 进程的相关术语
    - 作业、步骤、加载模块、任务、线程。

Basic Concepts

- **基本概念**
  - 一个进程包括一些段：
    - 文本：可执行（程序）代码
    - 数据与堆(Heap)
      - 数据：全局变量
      - 堆：在运行时动态分配的内存
    - 堆栈(Stack)
      - 临时数据存储
        - 过程/函数参数、返回地址、局部变量
  - 程序或进程的当前活动包括其上下文。
    - 程序计数器（PC）、处理器寄存器等。
  - 一个程序可以对应多个进程。
    - 执行同一顺序程序的多个用户
    - 运行多个进程的并发程序

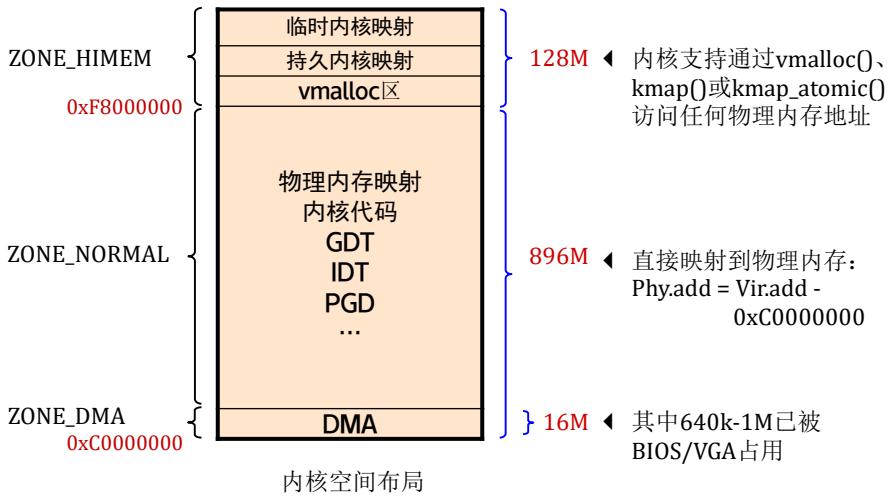
Basic Concepts

- **基本概念**
  - 进程虚拟内存—Linux/IA-32上的典型布局



Basic Concepts

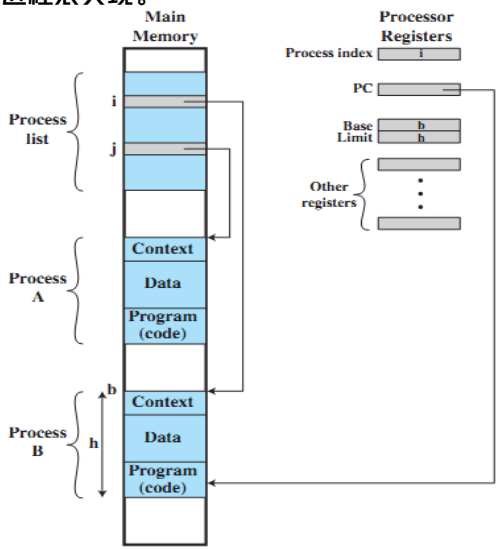
- **基本概念**
  - 进程虚拟内存—Linux/IA-32上的典型布局。



内核空间布局

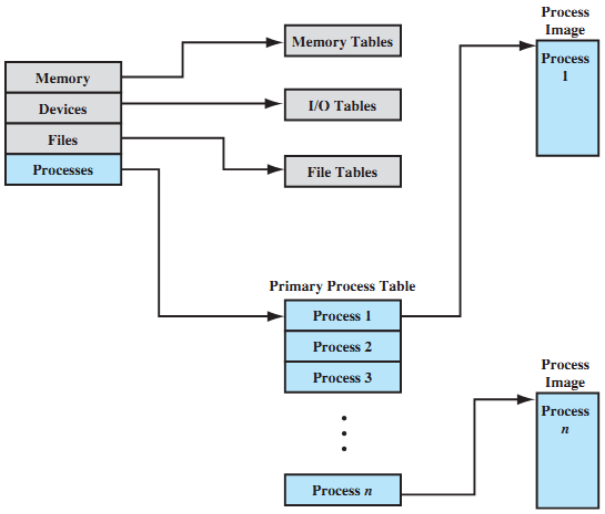
Basic Concepts

- 基本概念
  - 典型的进程表实现。



Basic Concepts

- 基本概念
  - 操作系统控制表的一般结构。



Basic Concepts

- **基本概念**
  - 进程属性
    - 进程ID
    - 父进程ID
    - 用户ID
    - 进程状态
    - 进程优先级
    - 程序计数器
    - 寄存器
    - 内存管理信息
    - I/O状态信息
    - 访问控制
    - 会计信息

Process Table and PCB

- **进程表**
  - 进程表是一种内核数据结构，包含内核必须始终可用的字段
    - 状态字段（标识进程状态）
    - 允许内核在内存中定位进程的字段
    - 确定各种进程权限的UID
    - 指定与进程的关系的PID（例如，fork）
    - 事件描述符（进程处于睡眠状态时）
    - 用于确定进程移动到“内核运行”和“用户运行”状态的顺序的调度参数
    - 发送到进程但尚未处理的信号的信号字段
    - 在内核模式和用户模式下提供进程执行时间的定时器
    - 提供进程大小的字段（以便内核知道为进程分配多少空间）

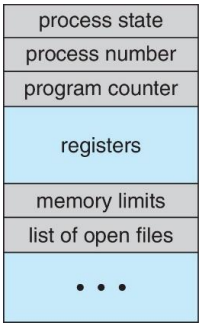
Process Table and PCB

- **进程表**
  - 典型进程表的字段。

进程管理	内存管理	文件管理
寄存器	指向文本段信息的指针	根目录
程序计数器	指向数据段信息的指针	工作目录
程序状态字	指向堆栈段信息的指针	文件描述符
堆栈指针		用户ID
进程状态		组ID
优先级		
调度参数		
进程ID		
父进程		
进程组		
信号		
进程启动的时间		
使用的CPU时间		
子进程的CPU时间		
下次警报时间		

Process Table and PCB

- **进程控制块（PCB）**
  - 每个进程在操作系统中由进程控制块（PCB）表示
    - PCB也称为任务控制块，IBM命名，表示与特定进程相关的信息
      - 它保存了进程的上下文
  - PCB是操作系统控制进程所需的数据（进程属性）：
    - 进程位置信息
    - 进程标识信息
    - 处理器状态信息
    - 进程控制信息



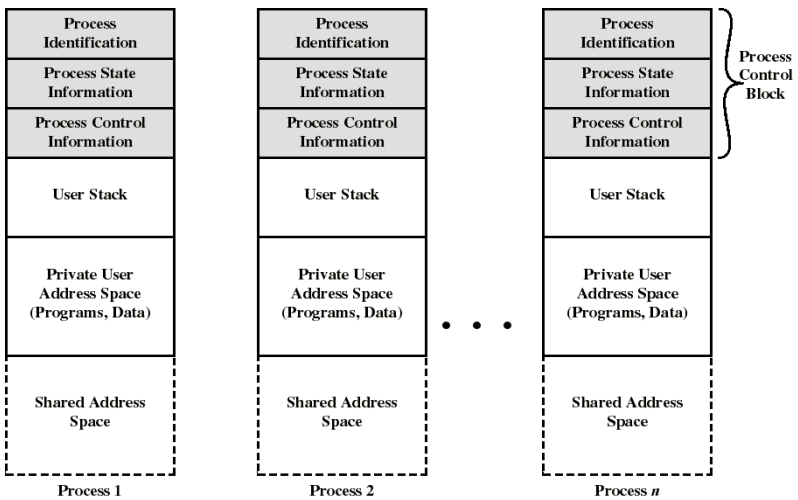
进程控制块（PCB）

Process Table and PCB

- **进程控制块（PCB）**
  - 进程位置信息
    - 进程映像(image)
      - 每个进程在内存中都有一个映像
      - 它可能不占用连续的地址范围
        - 取决于所使用的内存管理方案
      - 可以使用专用和共享内存地址空间。
    - 每个进程映像都由进程表中的一个条目指向。
    - 为了让操作系统管理进程，至少必须将其映像的一部分放入主内存。

Process Table and PCB

- **进程控制块（PCB）**
  - 进程位置信息。



内存中的进程映像

Process Table and PCB

- **进程控制块（PCB）**
  - 进程标识信息
    - 可以使用一些数字标识符：
      - 唯一进程标识符（PID）
        - 索引（直接或间接）到进程表中
      - 用户标识符（UID）
        - 负责该作业的用户
      - 创建此进程的进程的标识符（PPID、父进程ID）
    - 可能是与数字标识符相关的符号名

Process Table and PCB

- **进程控制块（PCB）**
  - 处理器状态信息
    - 处理器寄存器的内容
      - 用户可见寄存器
      - 控制和状态寄存器
      - 堆栈指针。
    - 程序状态字（PSW）
      - 包含状态信息
      - 例如
        - 奔腾计算机上的EFLAGS寄存器

Process Table and PCB

- **进程控制块（PCB）**
  - 进程控制信息
    - 调度和状态信息
      - 进程状态（例如，正在运行、准备就绪、已阻塞…）
      - 进程的优先级
      - 进程正在等待的事件（如果被阻塞）
    - 数据结构信息
      - 可能包含指向进程队列、父子关系和其他结构的其他PCB的指针
    - 进程间通信（IPC）
      - 可保存IPC的标志和信号
    - 资源所有权和利用
      - 正在使用的资源：打开文件、I/O设备……
      - 使用历史（CPU时间、I/O…）

Process Table and PCB

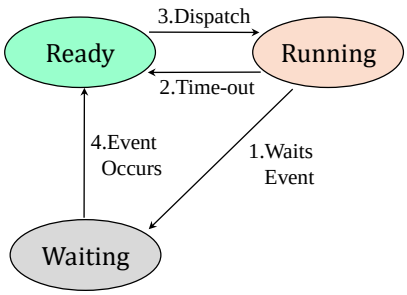
- **进程控制块（PCB）**
  - 进程控制信息（续）
    - 进程权限（访问控制）
      - 访问某些内存位置、资源等。
    - 内存管理
      - 指向分配给此进程的段/页表的指针
    - 程序计数器
      - 指示要为此进程执行的下一条指令的地址
    - 会计信息
      - 包括CPU和实时使用量、时间限制、帐号、作业或进程号等
    - I/O状态信息
      - 包括分配给进程的I/O设备列表、打开的文件列表等

Process States and Transitions

- **进程状态和转换**
  - 三态进程模型
    - 运行状态
      - 执行的进程；它的指令正在执行
    - 就绪状态
      - 准备好执行的任何进程；进程正在等待分配给处理器
    - 等待/阻塞状态
      - 在I/O完成或发生其他事件之前无法执行的任何进程

Process States and Transitions

- **进程状态和转换**
  - 三态进程模型。



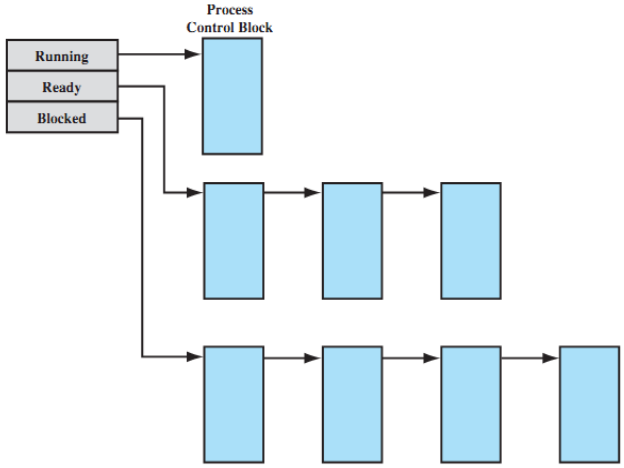
1. 进程阻塞等待输入
2. 调度程序选择另一个进程
3. 调度程序选择此进程
4. 输入变成可用

Process States and Transitions

- **进程状态和转换**
  - 三态进程模型
    - 进程状态转换
      - 就绪→运行：当时间到了，调度器选择一个新进程来运行
      - 运行→运行
        - 正在运行的进程已超时
        - 正在运行的进程被中断，因为优先级较高的进程处于就绪状态。
      - 运行→阻塞
        - 当进程请求它必须等待的内容时：
          - 要执行的操作系统服务尚未准备好
          - 要访问的资源还不可用
          - 启动I/O并等待结果
          - 等待其他进程提供输入
      - 阻塞→就绪：当进程等待的事件发生时

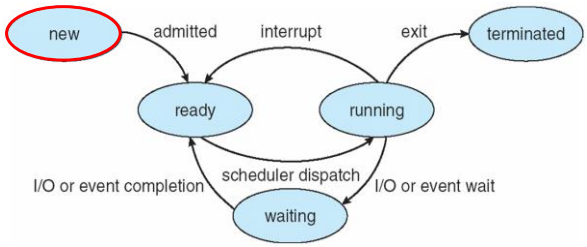
Process States and Transitions

- **进程状态和转换**
  - 三态进程模型
    - 进程列表结构。



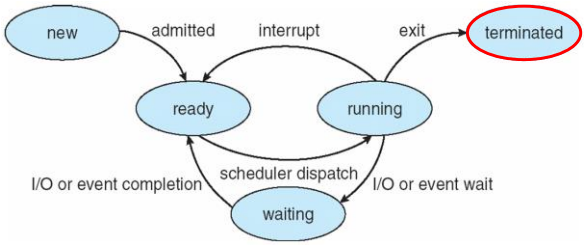
Process States and Transitions

- **进程状态和转换**
  - 五态进程模型
    - 新建状态
      - 操作系统已执行创建进程所需的操作：
        - 已创建进程标识符
        - 已创建管理进程所需的表
      - 但尚未确定执行该进程：
        - 因为资源有限



Process States and Transitions

- **进程状态和转换**
  - 五态进程模型
    - 终止状态
      - 进程终止将进程移动到终止状态。
        - 它不再有资格被执行。
      - 表格和其他信息临时保留，以供辅助程序使用。
        - 例如，累积资源使用以向用户计费的会计程序
      - 当不再需要数据时，进程（及其表）将被删除。



Operations on Processes

- **进程创建**
  - 何时创建进程
    - 系统初始化
    - 提交批处理作业
    - 用户登录
    - 由操作系统创建以向用户提供服务
      - 例如，打印文件
    - 用户请求创建新进程
    - 由现有进程产生(繁衍)
      - 一个程序可以指定（要求或必要时决定）许多进程的创建
      - 创建进程是父进程，创建的新进程称为该进程的子进程。

Operations on Processes

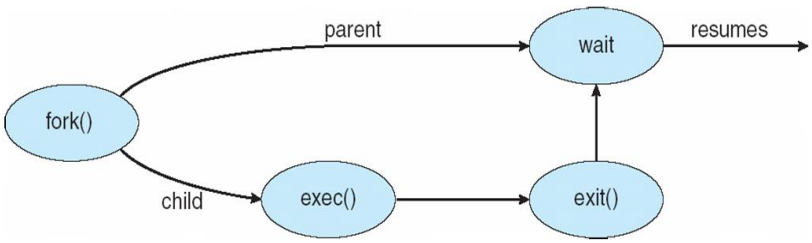
- **进程创建**
  - 创建进程中的详情
    - 父进程创建子进程，子进程又创建其他进程，形成进程树。
    - 可能的资源共享：
      - 父进程和子进程共享所有资源。
      - 子进程共享父进程资源的子集。
      - 父进程和子进程不共享任何资源。
    - 可能的执行：
      - 父进程和子进程同时执行。
      - 父进程等待子进程终止。

Operations on Processes

- **进程创建**
  - 创建进程中的详情（续）
    - 分配唯一的进程标识符（PID）。
      - 通常为整数
    - 为进程映像分配空间。
    - 初始化进程控制块（PCB）。
      - 许多默认值
        - 例如，状态是新的，没有I/O设备或文件。。。。
    - 建立适当的链接。
      - 例如，将新进程添加到用于调度队列的链表中
    - 地址空间
      - 子进程是父进程的副本，或
      - 子进程中加载了另一个程序

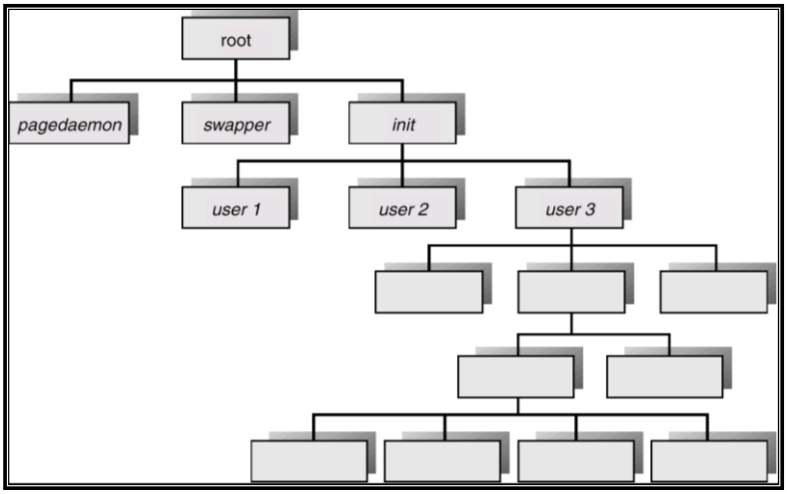
Operations on Processes

- **进程创建**
  - 创建进程中的详情（续）
    - UNIX示例
      - fork()系统调用创建新进程。
      - 在fork()之后使用exec()系统调用，用新程序替换进程的内存空间。



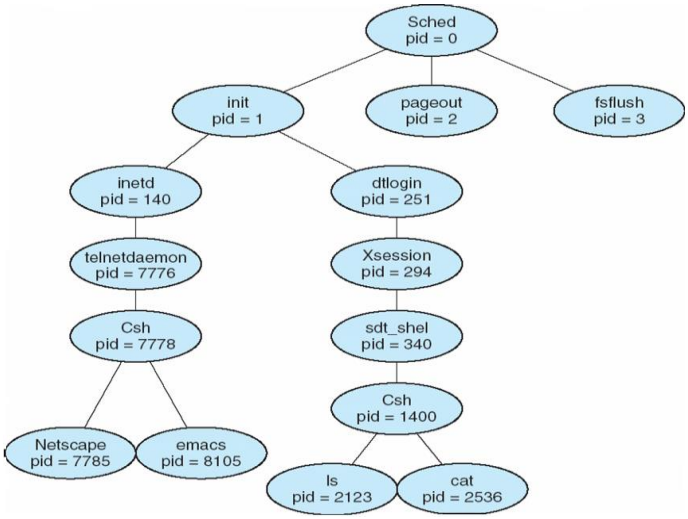
Operations on Processes

- **进程创建**
  - UNIX上的进程树。



Operations on Processes

- **进程创建**
  - 典型Solaris上的进程树。



Operations on Processes

- **进程创建**
  - 算法6-1：fork-demo.c（复刻一个单独的进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int count = 1;
    pid_t childpid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    } else /* fork() 在产生的子进程中，返回值是0 */
        if (childpid == 0) { /* 子进程 */
            count++;
            printf("子进程ID=%d, count=%d(地址=%p)\n", getpid(), count, &count);
        } else { /* 父进程 */
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d\n", childpid);
            wait(0); /* 等待所有子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-1：fork-demo.c（复刻一个单独的进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int count = 1;
    pid_t childpid;
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-1-fork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=574, count=1(地址=0x7ffd601ec860) childpid=575
子进程ID=575, count=2(地址=0x7ffd601ec860)
进程575到达检测点
进程574到达检测点
```

```c
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d\n", childpid);
            wait(0); /* 等待所有子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-1：fork-demo.c（复刻一个单独的进程）

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int count = 1;
    pid_t childpid;
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-1-fork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=574, count=1(地址=0x7ffd601ec860) childpid=575
子进程ID=575, count=2(地址=0x7ffd601ec860)
进程575到达检测点
进程574到达检测点
```

子进程中的变量count与父进程中的变量count具有相同的虚拟地址。

```
            printf(" childpid=%d\n", childpid);
            wait(0); /* 等待所有子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-1：fork-demo.c（复刻一个单独的进程）

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int count = 1;
    pid_t childpid;
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-1-fork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=574, count=1(地址=0x7ffd601ec860) childpid=575
子进程ID=575, count=2(地址=0x7ffd601ec860)
进程575到达检测点
进程574到达检测点
```

子进程中的count值与父进程中的count值不同。它们被映射到不同进程映像中的不同物理地址。

```
            wait(0); /* 等待所有子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-1：fork-demo.c（复刻一个单独的进程）

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int count = 1;
    pid_t childpid;
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-1-fork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=574, count=1(地址=0x7ffd601ec860) childpid=575
子进程ID=575, count=2(地址=0x7ffd601ec860)
进程575到达检测点
进程574到达检测点
```

父子进程都执行了检测点输出。

```
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d\n", childpid);
            wait(0); /* 等待所有子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        return EXIT_FAILURE;
    } else /* vfork() 在产生的子进程中，返回值是0 */
        if (childpid == 0) { /* 子进程 */
            count++;
            printf("子进程ID=%d, count=%d(地址=%p)\n", getpid(), count, &count);
            printf("子进程睡着了……\n");
            sleep(10);
            printf("子进程醒来了!\n");
            _exit(0); /* 或exec(0); "return"将造成栈错误 */
        } else { /* 父进程，子进程结束后才会开始执行 */
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d\n", childpid);
            wait(0); /* 无需等待子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
        printf(" childpid=%d\n", childpid);
        wait(0); /* 无需等待子进程结束 */
    }
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-2-vfork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=622, count=2(地址=0x7fff33b24280)
子进程睡着了……
子进程醒来了!
父进程ID=621, count=2(地址=0x7fff33b24280) childpid=622
进程621到达检测点
```

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        printf("父进程ID=%d, cou
        printf(" childpid=%d\n
        wait(0); /* 无需等待子进
    }
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-2-vfork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=622, count=2(地址=0x7fff33b24280)
子进程睡着了……
子进程醒来了!
父进程ID=621, count=2(地址=0x7fff33b24280) childpid=622
进程621到达检测点
```

子进程中的变量count与父进程中的变量count具有相同的虚拟地址。

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        printf("父进程ID=%d, count
        printf(" childpid=%d\n",
        wait(0); /* 无需等待子进程结
    }
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-2-vfork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=622, count=2(地址=0x7fff33b24280)
子进程睡着了……
子进程醒来了!
父进程ID=621, count=2(地址=0x7fff33b24280) childpid=622
进程621到达检测点
```

子进程中count的值与父进程中count的值相同。它们被映射到同一进程映像中的同一物理地址。

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        printf("父进程ID=
        printf(" childpi
        wait(0); /* 无需等待子进程结束 */
    }
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-2-vfork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=622, count=2(地址=0x7fff33b24280)
子进程睡着了……
子进程醒来了!
父进程ID=621, count=2(地址=0x7fff33b24280) childpid=622
进程621到达检测点
```

父进程将挂起，直到vforked子进程终止才继续执行。

Operations on Processes

- **进程创建**
  - 算法6-2：vfork-demo.c（复刻一个共享空间进程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        exit(EXIT_FAILURE);
    } else if (childpid == 0) {
        printf("子进程ID=%d, count=%d(地址=%p)\n", getpid(), ++count, &count);
        printf("子进程睡着了……\n");
        sleep(3);
        printf("子进程醒来了!\n");
        exit(EXIT_SUCCESS);
    }
    printf("父进程ID=%d, count=%d(地址=%p) childpid=%d\n", getpid(), count, &count, childpid);
    printf("进程%d到达检测点\n", getpid()); /* 仅父进程到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-2-vfork-demo.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=622, count=2(地址=0x7fff33b24280)
子进程睡着了……
子进程醒来了!
父进程ID=621, count=2(地址=0x7fff33b24280) childpid=622
进程621到达检测点
```

子进程在测试点之前退出了，
仅父进程执行到检测点

Operations on Processes

- **进程终止**
  - 当下列事件之一发生时，进程终止
    - 批处理作业发出Halt指令
    - 用户注销
    - 进程执行服务请求以终止
    - 父进程终止子进程
    - 错误和故障发生

Operations on Processes

- **进程终止**
  - 进程终止的原因
    - 正常/错误/致命退出
    - 超过时限
    - 时间超限：进程等待事件的时间超过了指定的最大值
    - 内存不可用
    - 内存访问越界
    - 保护错误：例如，写入只读文件
    - 算术错误
    - I/O故障
    - 无效指令：在尝试执行数据时发生
    - 特权指令
    - 操作系统介入：例如死锁发生时
    - 父进程要求终止一个子进程
    - 父进程终止，子进程也终止

Operations on Processes

- **进程终止**
  - 进程终止的过程
    - 进程可以执行完最后一条语句，并通过exit()系统调用请求操作系统终止它。
      - 它在进程表中的条目将一直保留，直到其父进程（如果存在）调用wait()
      - 它的资源由操作系统分配。
    - 父进程可以终止子进程的执行：
      - 子进程已超出分配的资源
      - 任务不再需要分配给子进程执行
      - 如果父进程正在退出：
        - 某些操作系统不允许子进程在父进程终止时继续运行
        - 级联终止 – 所有的子进程都终止了

Operations on Processes

- **进程终止**
  - 进程终止的过程
    - wait()系统调用

```
#include <sys/wait.h>
/* pid_t wait(int *status); */

pid_t pid;
int status;
pid = wait(&status); /* 等待子进程结束，
                        返回子进程的status与pid */
```

    - 当进程终止时，操作系统将释放其资源。但是，在父进程调用wait()之前，它在进程表中的条目必须保留在那里，因为进程表包含进程的退出状态。

Operations on Processes

- **进程终止**
  - 僵尸和孤儿
    - 已终止但其父进程尚未调用wait()的进程已失效，称为僵尸进程(zombies).
      - 所有进程在终止时都会转换到此状态，但通常它们只是短暂地作为僵尸存在。父进程调用wait()后，将释放僵尸进程的进程标识符及其在进程表中的条目。
    - 现在考虑一下如果一个父进程没有调用wait()而终止，那么会把它的子进程作为孤儿进程(orphans)留下来
    - Linux和UNIX通过将init进程指定为孤儿进程的新父进程（收养孤儿进程）来解决这种情况。
      - init进程是UNIX和Linux系统中进程层次结构的根。
      - init进程定期调用wait()，从而允许收集任何孤立进程的退出状态，并释放孤立进程的进程标识符和进程表条目。

Operations on Processes

- **进程终止**
  - 算法6-3：fork-demo-nowait.c（不等待复刻子进程的终止）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    } else /* fork() 在产生的子进程中，返回值是0 */
        if (childpid == 0) { /* 子进程 */
            count++;
            printf("子进程ID=%d, count=%d(地址=%p)\n", getpid(), count, &count);
            printf("子进程睡着了……\n");
            sleep(10);
            printf("\n子进程醒来了！\n");
        } else { /* 父进程 */
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d\n", childpid);
            // wait(0); /* 不等待子进程结束 */
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程终止**
  - 算法6-3：fork-demo-nowait.c（不等待复刻子进程的终止）

```
lab@ByteDanceServer:~/os$ gcc alg.6-3-fork-demo-nowait.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=1673, count=1(地址=0x7fffa2d24ba0) childpid=1674
进程1673到达检测点
子进程ID=1674, count=2(地址=0x7fffa2d24ba0)
子进程睡着了……
lab@ByteDanceServer:~/os$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    18    17  0  80   0 -  2577 do_wai pts/0    00:00:01 bash
3 S  1000    97    17  0  80   0 -  1759 core_s pts/0    00:00:00 dbus-launch
1 S  1000  1674    17  0  80   0 -   623 hrtime pts/0    00:00:00 a.out
0 R  1000  1675    18  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os$
子进程醒来了！
进程1674到达检测点
```

Operations on Processes

- **进程终止**
  - 算法6-3：fork-demo-nowait.c（不等待复刻子进程的终止）

```
lab@ByteDanceServer:~/os$ gcc alg.6-3-fork-demo-nowait.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=1673, count=1(地址=0x7fffa2d24ba0) childpid=1674
进程1673到达检测点
子进程ID=1674, count=2(地址=0x7fffa2d24ba0)
子进程睡着了……
lab@ByteDanceServer:~/os$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    18    17  0  80   0 -  2577 do_wai pts/0    00:00:01 bash
3 S  1000    97    17  0  80   0 -  1759 core_s pts/0    00:00:00 dbus-launch
1 S  1000  1674    17  0  80   0 -   623 hrtime pts/0    00:00:00 a.out
0 R  1000  1675    18  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os$
子进程醒来了!
进程1674到达检测点
```

父进程终止，只剩下一个孤立的子进程 pid=1674。

```
#include <stdio.h>
int main(void){
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程终止**
  - 算法6-3：fork-demo-nowait.c（不等待复刻子进程的终止）

```
lab@ByteDanceServer:~/os$ gcc alg.6-3-fork-demo-nowait.c
lab@ByteDanceServer:~/os$ ./a.out
父进程ID=1673, count=1(地址=0x7fffa2d24ba0) childpid=1674
进程1673到达检测点
子进程ID=1674, count=2(地址=0x7fffa2d24ba0)
子进程睡着了……
lab@ByteDanceServer:~/os$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    18    17  0  80   0 -  2577 do_wai pts/0    00:00:01 bash
3 S  1000    97    17  0  80   0 -  1759 core_s pts/0    00:00:00 dbus-launch
1 S  1000  1674    17  0  80   0 -   623 hrtime pts/0    00:00:00 a.out
0 R  1000  1675    18  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os$
子进程醒来了!
进程1674到达检测点
```

这里发生了什么？终端（bash）和分支子进程正在异步执行

```
#include <stdio.h>
int main(void){
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程终止**
  - 算法6-4：fork-demo-wait.c（等待复刻子进程的终止）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
int main(void){
    int count = 1;
    pid_t childpid, terminatedid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
        return EXIT_FAILURE;
    } else /* fork() 在产生的子进程中，返回值是0 */
        if (childpid == 0) {  /* 子进程 */
            count++;
            printf("子进程ID=%d, count=%d(地址=%p)\n", getpid(), count, &count);
            printf("子进程睡着了……\n");
            sleep(10);
            printf("\n子进程醒来了！\n");
        } else {              /* 父进程 */
            terminatedid = wait(0); /* 等待子进程结束 */
            printf("父进程ID=%d, count=%d(地址=%p)", getpid(), count, &count);
            printf(" childpid=%d, terminatedid=%d\n", childpid, terminatedid);
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程终止**
  - 算法6-4：fork-demo-wait.c（等待复刻子进程的终止）


```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```


```c
int main(void){
    int count = 1;
    pid_t childpid, terminatedid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-4-fork-demo-wait.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=1802, count=2(地址=0x7ffd9a53ae3c)
子进程睡着了……

子进程醒来了！
进程1802到达检测点
父进程ID=1801, count=1(地址=0x7ffd9a53ae3c) childpid=1802, terminatedid=1802
进程1801到达检测点
lab@ByteDanceServer:~/os$
```

```c
            printf(" childpid=%d, terminatedid=%d\n", childpid, terminatedid);
        }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

Operations on Processes

- **进程终止**
  - 算法6-4：fork-demo-wait.c（等待复刻子进程的终止）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid, terminatedid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
        printf(" childpid=%d, terminatedid=%d\n", childpid, terminatedid);
    }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-4-fork-demo-wait.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=1802, count=2(地址=0x7ffd9a53ae3c)
子进程睡着了……

子进程醒来了！
进程1802到达检测点
父进程ID=1801, count=1(地址=0x7ffd9a53ae3c) childpid=1802, terminatedid=1802
进程1801到达检测点
lab@ByteDanceServer:~/os$
```

父进程wait()一直等待，直到子进程终止才继续执行。

Operations on Processes

- **进程终止**
  - 算法6-4：fork-demo-wait.c（等待复刻子进程的终止）

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void){
    int count = 1;
    pid_t childpid, terminatedid;

    childpid = fork(); /* 子进程复制父进程地址空间 */
    if (childpid < 0) {
        perror("fork()");
        printf(" childpid=%d, terminatedid=%d\n", childpid, terminatedid);
    }
    printf("进程%d到达检测点\n", getpid()); /* 子进程先到达检测点 */
    return EXIT_SUCCESS;
}
```

```
lab@ByteDanceServer:~/os$ gcc alg.6-4-fork-demo-wait.c
lab@ByteDanceServer:~/os$ ./a.out
子进程ID=1802, count=2(地址=0x7ffd9a53ae3c)
子进程睡着了……

子进程醒来了！
进程1802到达检测点
父进程ID=1801, count=1(地址=0x7ffd9a53ae3c) childpid=1802, terminatedid=1802
进程1801到达检测点
lab@ByteDanceServer:~/os$
```

子进程先到达检测点，然后父进程再到达。

Operations on Processes

- **进程终止**
  - 算法6-5-0：sleeper.c（休眠5秒的演示进程）

```c
/* gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    int second = 5;
    if (argc > 1) {
        second = atoi(argv[1]);
        if (second <= 0 || second > 10)
            second = 5;
    }
    /* ppid -父进程的PID */
    printf("\n休眠进程pid=%d, ppid=%d\n", getpid(), getppid());
    printf("休眠%d秒...\n", second);
    sleep(second);
    printf("\n进程醒来并返回.\n");
    return 0;
}
```

Operations on Processes

- **进程终止**
  - 算法6-5：vfork-exec-wait.c（复刻、执行和等待）

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <wait.h>
```

```c
int main(int argc, char *argv[]){
    pid_t childpid;
    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        return EXIT_FAILURE;
    } else if (childpid == 0) { /* 这是子进程 */
        printf("子进程pid=%d, 休眠2秒...\n", getpid());
        sleep(2); /* 此时父进程挂起 */
        char filename[80];
        struct stat buf;
        strcpy(filename, "./alg.6-5-0-sleeper.o");
        if (stat(filename, &buf) == -1) {
            perror("\n休眠进程stat()");
            _exit(0); }
        char *argv1[] = {filename, argv[1], NULL};
        printf("子进程醒了，execv()执行休眠进程:%s %s\n\n", argv1[0], argv1[1]);
        execv(filename, argv1); /* 'execv'调用时父进程恢复执行. */
    } else  { /* 这是父进程，当vfork的子进程终止时开始执行，即execv执行时执行。 */
        printf("父进程pid=%d, childpid=%d\n", getpid(), childpid);
        int retpid = wait(0); /* 如果没有wait(), execv子进程将成为孤儿 */
        printf("\nwait()返回childpid=%d\n", retpid);
    }
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-5：vfork-exec-wait.c（复刻、执行和等待）

```
lab@ByteDanceServer:~/os$ gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c
lab@ByteDanceServer:~/os$ ./alg.6-5-0-sleeper.o

休眠进程pid=2630, ppid=18
休眠5秒...

进程醒来并返回.
lab@ByteDanceServer:~/os$ gcc alg.6-5-vfork-execv-wait.c
lab@ByteDanceServer:~/os$ ./a.out 6
子进程pid=2646, 休眠2秒...
子进程醒了, execv()执行休眠进程:./alg.6-5-0-sleeper.o 6

父进程pid=2645, childpid=2646

休眠进程pid=2646, ppid=2645
休眠6秒...

进程醒来并返回.

wait()返回childpid=2646
lab@ByteDanceServer:~/os$
```

```
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-5：vfork-exec-wait.c（复刻、执行和等待）

```
lab@ByteDanceServer:~/os$ gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c
lab@ByteDanceServer:~/os$ ./alg.6-5-0-sleeper.o

休眠进程pid=2630, ppid=18
休眠5秒...

进程醒来并返回.
lab@ByteDanceServer:~/os$ gcc alg.6-5-vfork-execv-wait.c
lab@ByteDanceServer:~/os$ ./a.out 6
子进程pid=2646, 休眠2秒...
子进程醒了, execv()执行休眠进程:./alg.6-5-0-sleeper.o 6

父进程pid=2645, childpid=2646

休眠进程pid=2646, ppid=2645
休眠6秒...

进程醒来并返回.

wait()返回childpid=2646
lab@ByteDanceServer:~/os$
```

休眠进程继承了分支子进程的pid（2646）

```
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-5：vfork-exec-wait.c（复刻、执行和等待）

```
lab@ByteDanceServer:~/os$ gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c
lab@ByteDanceServer:~/os$ ./alg.6-5-0-sleeper.o

休眠进程pid=2630, ppid=18
休眠5秒...

进程醒来并返回.
lab@ByteDanceServer:~/os$ gcc alg.6-5-vfork-execv-wait.c
lab@ByteDanceServer:~/os$ ./a.out 6
子进程pid=2646, 休眠2秒...
子进程醒了, execv()执行休眠进程:./alg.6-5-0-sleeper.o 6

父进程pid=2645, childpid=2646

休眠进程pid=2646, ppid=2645
休眠6秒...

进程醒来并返回.

wait()返回childpid=2646
lab@ByteDanceServer:~/os$
```

父进程在分支子进程调用execv时恢复执行，分支子进程在该点终止，休眠进程sleeper作为子进程生成在同一个childpid中，但具有复制的地址空间，返回父进程时不会造成堆栈崩溃。父、子进程异步执行。

```
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-5：vfork-exec-wait.c（复刻、执行和等待）

```
lab@ByteDanceServer:~/os$ gcc -o alg.6-5-0-sleeper.o alg.6-5-0-sleeper.c
lab@ByteDanceServer:~/os$ ./alg.6-5-0-sleeper.o

休眠进程pid=2630, ppid=18
休眠5秒...

进程醒来并返回.
lab@ByteDanceServer:~/os$ gcc alg.6-5-vfork-execv-wait.c
lab@ByteDanceServer:~/os$ ./a.out 6
子进程pid=2646, 休眠2秒...
子进程醒了, execv()执行休眠进程:./alg.6-5-0-sleeper.o 6

父进程pid=2645, childpid=2646

休眠进程pid=2646, ppid=2645
休眠6秒...

进程醒来并返回.

wait()返回childpid=2646
lab@ByteDanceServer:~/os$
```

不过，父进程需要wait子进程，否则产生的睡眠者子进程将会成为孤儿

```
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <wait.h>

int main(int argc, char *argv[]){
    pid_t childpid;
    childpid = vfork(); /* 子进程共享父进程地址空间 */
    if (childpid < 0) {
        perror("vfork()");
        return EXIT_FAILURE;
    } else if (childpid == 0) { /* 这是子进程 */
        printf("子进程pid=%d, 休眠2秒...\n", getpid());
        sleep(2); /* 此时父进程挂起 */
        char filename[80];
        struct stat buf;
        strcpy(filename, "./alg.6-5-0-sleeper.o");
        if (stat(filename, &buf) == -1) {
            perror("\n休眠进程stat()");
            _exit(0); }
        char *argv1[] = {filename, argv[1], NULL};
        printf("子进程醒了, execv()执行休眠进程:%s %s\n\n", argv1[0], argv1[1]);
        execv(filename, argv1); /* 'execv'调用时父进程恢复执行. */
    } else  { /* 这是父进程, 当vfork的子进程终止时开始执行, 即execv执行时执行。 */
        printf("父进程pid=%d, childpid=%d\n", getpid(), childpid);
        printf("父进程调用shell的ps\n");system("ps -l");sleep(1);
        return EXIT_SUCCESS; /* 没有wait(), 子进程将成为孤儿 */
    }
    return EXIT_SUCCESS;}
```

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:./alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

bash是a.out的父进程

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

a.out是sleeper的父进程，它从execv()继承了vorked子进程的pid

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

a.out是system()命令产生的sh进程的父进程

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

sh是ps的父进程，ps由命令产生：system("ps -l")

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

a.out终止，控制返回bash终端，从终端键入“ps -l”

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

PPID的变化表明睡眠者失去他的父母后被11331收养了

pid 11331是init进程，OS的根进程

Operations on Processes

- **进程终止**
  - 算法6-6:vfork-execv-nowait.c （复刻、执行、不等待）

```
lab@ByteDanceServer:~/os/lec06$ gcc alg.6-6-vfork-execv-nowait.c
lab@ByteDanceServer:~/os/lec06$ ./a.out 6
子进程pid=15175, 休眠2秒...
子进程醒了, execv()执行休眠进程:../alg.6-5-0-sleeper.o 6

父进程pid=15174, childpid=15175
父进程调用shell的ps

休眠进程pid=15175, ppid=15174
休眠6秒...
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15174 11332  0  80   0 -   623 do_wai pts/0    00:00:00 a.out
0 S  1000 15175 15174  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 S  1000 15176 15174  0  80   0 -   653 do_wai pts/0    00:00:00 sh
0 R  1000 15177 15176  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000 11332 11331  0  80   0 -  2579 do_wai pts/0    00:00:00 bash
0 S  1000 15175 11331  0  80   0 -   623 hrtime pts/0    00:00:00 alg.6-5-0-sleep
0 R  1000 15178 11332  0  80   0 -  2635 -      pts/0    00:00:00 ps
lab@ByteDanceServer:~/os/lec06$
进程醒来并返回.
lab@ByteDanceServer:~/os/lec06$ ps -q 11331
  PID TTY          TIME CMD
11331 ?        00:00:00 init
```

终端（bash）和睡眠者进程异步执行