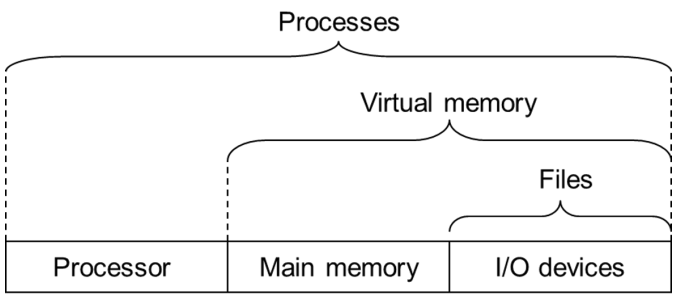
# Process



## 一、概念

### 1. 抽象




### 2. Exception

控制流中的突变

**类型**

- 异步
  - I/O interrupt

    ctrl+c；网络包
  - 硬重置：power off
  - 软重置：ctrl-alt-delete
- 同步
  - Trap
  - Fault
  - Abort

**处理：context switch**

对CPU的分时共享（并发）

time slice：被动切换。有timer的硬件，定时给CPU发中断信号

block：主动。

**进程**

PID：唯一标识符。PID > 0

Fork：创建子进程。返回两次。

子进程：与父进程一样的内存空间、用户栈等，是一份拷贝的副本，之后不相互影响。但是**文件标识符**是共享的，通过文件传递信息可以实现父子交互。

```
// typedef pid_t int
pid_t getpid(void); // 当前pid
pid_t getppid(void);  //父进程pid
void exit(int status); //进程显式终止
pid_t fork(void); //子进程返回0，父进程返回子进程的pid，出错-1
```

Reap：系统init（pid=1）回收未回收的zombie。

wait：父进程显示回收

```
// 成功返回子进程pid，如果WNOHANG且无子进程，返回0；其他错误-1
pid_t waitpid(pid_t pid, int *statusp, int options);
// 等价于waitpid(-1, &status,0)
pid_t wait(int *statusp);

while(waitpid(-1, &status, 0) > 0) {
    // 这个循环会回收所有子进程，直到没有子进程才返回0，退出while
}
```

- pid：pid>0为子进程号码；-1为所有子进程
- options
  - 0：等待集合中一个子进程终止就返回。
  - WNONHANG：看一眼，没有子进程终止就返回，不会一直等
  - WUNTRACED：也会返回被停止的pid
  - WCONTINUED：也会返回因SIGCONT停止的pid
  - WNONHANG | WUNTRACED：立即返回，检查有无终止或被停止的
- statusp：填写子进程退出状态p745

```
unsigned int sleep(unsigned int secs); //将进程挂起指定的时间
int pause(void); //休眠直至收到信号
int execve(const char *filename, const char *argv[], const char *envp[]); //成功不返回，错误-1

```

## 二、习题

## 1. fork

```c
void handler(int sig) {
   static int beeps = 0;
   printf("BEEP\n");
   if (beeps < 4) {
      beeps += 1;
      fork();
      /* The next SIGALRM will be delivered in 1 second */
      alarm(1);
   } else {
      printf("BOOM!\n");
      exit(0);
   }
}

int main() {
   /* Install the SIGALRM handler */
   signal(SIGALRM, handler);

   /* The next SIGALRM will be delivered in 1 second */
   alarm(1);

   /* The control returns here each time after the signal handler */
   while (1);

   exit(0);
}
```

主进程4次BEEP后BOOM并退出，留下一堆僵尸子进程。

输出？$1+2+4+8+16=31$ 次BEEP，16次BOOM。（父子共享beeps）

## 2. Expection

Please specify which kind of exception(Faults, Aborts, Traps, Interrupts) will occur in the given scenario, point out whether it is asynchronous or synchronous, and specify where the exception handler will return to.

1. You dereference a NULL pointer.

   **Faults synchronous aborts the interrupted program**

2. The memory of your PC corrupted

   **Aborts synchronous aborts the interrupted program**

3. You run a command “kill -9 ¡pid¿” in your shell

   **Traps (System calls) synchronous returns control to the next instruction $I_{next}$**

4. You click your mouse

   **Interrupts asynchronous returns control to the next instruction $I_{next}$**

## 3. Context Switch

Consider the scenario where you are coding the lab. You have thought for a while without operating on your editor. After a key is pressed on keyboard, it displays in your screen. You can view this procedure as the combination of executing 'getchar' and 'printf'. What are the system calls the two functions finally issue? Can you describe how control flow switches between user mode and kernel mode in this procedure? SOLUTION: **The 'getchar' function finally issues the 'read' system call. The 'printf' function finally issues the 'write' system call. At first, the input monitoring thread calles 'getchar' in user mode, and then it switches to kernel mode to read user input. Since no input is received(without operating), the thread is put to sleep by kernel, and another thread is scheduled. (kernel mode to user mode) After a key is pressed on keyboard, an interrupt is sent and control switches to kernel mode. The monitoring thread is woken up, and gets the input char. (kernel mode to user mode) Then, the monitoring thread calles "printf" in user mode, and control switches to kernel mode. Kernel writes the char through graphics driver, then you can see it on your screen.**

## 4. Zombie

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main(void) {
8      if (fork() == 0) {
9          if (fork() == 0) { // Proc 1
10             exit(0);
11         } else { // Proc 2
12             waitpid(-1, NULL, 0);
13             exit(0);
14         }
15     } else { // Proc 3
16         while (waitpid(-1, NULL, WNOHANG) > 0);
17         exit(0);
18     }
19     return 0;
20 }
```

1. How many zombie processes could the program leave for init to reap? Please show all possible number and show one execution order for each. (Assume that shell will not reap the main process)

   **Since Proc 2 will wait for Proc 1 to finish, Proc 1 will not be reaped by init. Thus, we only consider the possible interleaving between Proc 2 and Proc3.**

   **There are two possible results:**

   **When Proc 2 exit before Proc 3 calling waitpid, only the Proc 3 will be reaped by init.**

   **When Proc 3 call waitpid before Proc 2 calling exit, both Proc 3 and Proc 2 will be reaped by init.**

2. What if we remove the line 12?

   **There are two possible results:**

   **Proc 1 and Proc 3 will be reaped by init.**

   **All three processes will be reaped by init.**