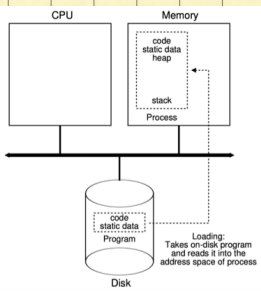
Process

- Process is a program in execution
- A program is a file on the disk
  - Code and static data
- A process is loaded by the OS
  - Code and static data are loaded from the program
  - Heap and stack are created by the OS

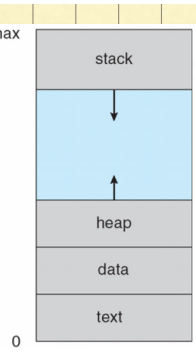


进程是正在执行的程序

静态：程序（存储在硬盘上）

动态：进程（OS创建堆、栈）

- A process is an abstraction of machine states
  - Memory: address space
  - Register:
    - Program Counter (PC) or Instruction Pointer
    - Stack pointer
    - frame pointer
  - I/O: all files opened by the process



进程是机器状态的抽象.

Process Identification

- How can we distinguish processes from one to another?
  - Each process is given a unique ID number, and is called the process ID, or the PID.
  - The system call, getpid(), prints the PID of the calling process.

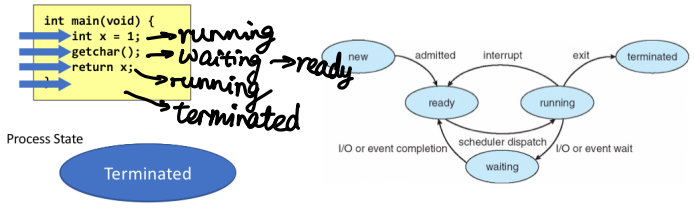
```
// compile to getpid
#include <stdio.h>   // printf()
#include <unistd.h>  // getpid()

int main(void) {
    printf("My PID is %d\n", getpid() );
}
```

```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

每个进程都有独一的ID.

PID会不断累加，满时会重新排.

Process Life Cycle



Waiting不能直接返回running

waiting也称block（等待、阻塞）

System Call : Process-Kernel Interaction

- System call is a function call.
  - exposed by the **kernel**.
  - abstraction of kernel operations.

```
int add_function(int a, int b) {
    return (a + b);
}

int main(void) {
   int result;
   result = add_function(a,b);
   return 0;
}

// this is a dummy example…
```

This is a function call.

```
int main(void) {
    time(NULL);
    return 0;
}
```

Process

system call

```
//somewhere in the kernel.
int time ( time_t * t ) {
    ......
    ret t;
}
```

Kernel

Here contains codes that access the hardware clock!

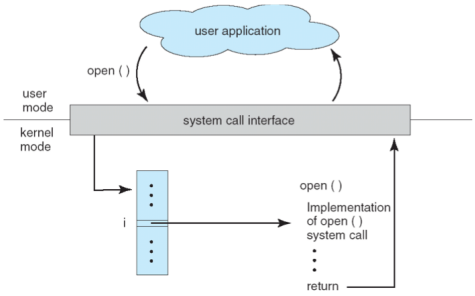
System call是函数调用

但是System call之后不会直接跳转到内核态，而是根据一张表，到内核态调用对应的system call.

System Call : Call by number

有trap table，根据数字去进行System Call

- System call is different from function call
- System call is a call by number



- User-mode code from xv6-riscv

```
int main(void) {
    ……
    int fd = open("copyin1", O_CREATE|O_WRONLY);
    ……
    return 0;
}
```

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* user/usys.S */
.global open
open:
    li a7, SYS_open
    ecall
    ret
```

- Kernel code from xv6-riscv

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* kernel/file.c */

uint64 sys_open(void) {
    ……
    return fd;
}
```

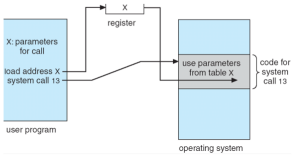
```
/* kernel/syscall.c */

static uint64 (*syscalls[])(void) = {
    .......
    [SYS_open] sys_open,
    .......
}

void syscall(void) {
    struct proc *p = myproc();
    num = p->trapframe->a7;
    p->trapframe->a0 = syscalls[num]();
}
```

System Call : Parameter Passing

传递参数方式：
寄存器
内存块
栈

- Often, more information is required than the index of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - **Registers**: pass the parameters in registers
    - In some cases, may be more parameters than registers
    - x86 and risc-v take this approach
  - **Blocks**: Parameters stored in a memory block and address of the block passed as a parameter in a register
  - **Stack**: Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
    - Block and stack methods do not limit the number or length of parameters being passed

- Example: parameter passing via blocks

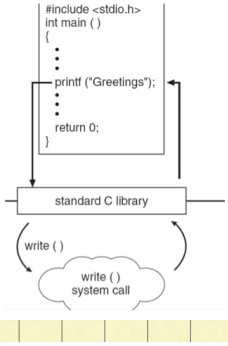


System Call vs. Library API Call

API不是系统调用

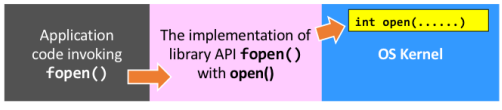
- Most operating systems provide standard C library to provide library API calls
  - A layer of indirection for system calls

Name	System call?
printf() & scanf()	No
malloc() & free()	No
fopen() & fclose()	No
mkdir() & rmdir()	Yes
chown() & chmod()	Yes



- Take **fopen()** as an example.
  - **fopen()** invokes the system call **open()**.
  - **open()** is too primitive and is not programmer-friendly!

Library call	fopen("hello.txt", "w");
System call	open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);

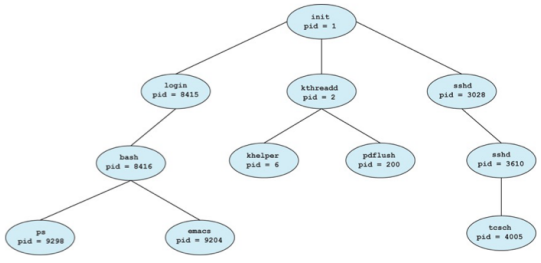


Process Creation

- 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
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

进程创建会形成进程树.
进程由pid唯一标识.
资源共享
并发执行
地址空间

- A tree of processes in Linux



Creating Processes with fork() System Call

```
$ ./fork_example_1
Ready (PID=1234)
My PID is 1234
My PID is 1235
$ _
```

PID 1234

Process 1234 is the original process, and we call it the parent process.

PID 1235

```
int main(void) {
    printf("Ready (PID = %d)\n", getpid());
    fork();
    printf("My PID is %d\n", getpid() );
    return 0;
}
```

Process 1235 is created by the **fork()** system call, and we call it the child process.

What do we know so far?

-Both the parent and the child execute **the same program**.

-The child process starts its execution **at the location that fork() is returned,** *not from the beginning of the program.*

父子进程执行相同的程序
子进程从fork()返回处开始执行.

```
1  int main(void) {
2    int result;
3    printf("before fork ...\n");
4    result = fork();
5    printf("result = %d.\n", result);
6
7    if(result == 0) {
8      printf("I'm the child.\n");
9      printf("My PID is %d\n", getpid());
10   }
11   else {
12     printf("I'm the parent.\n");
13     printf("My PID is %d\n", getpid());
14   }
15
16   printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
```

Important

- Both parent and child need to return from fork().
- CPU scheduler decides which to run first.

fork()返回后由CPU决定谁先运行.

```
1  int main(void) {
2    int result;
3    printf("before fork ...\n");
4    result = fork();
5    printf("result = %d.\n", result);
6
7    if(result == 0) {
8      printf("I'm the child.\n");
9      printf("My PID is %d\n", getpid());
10   }
11   else {
12     printf("I'm the parent.\n");
13     printf("My PID is %d\n", getpid());
14   }
15
16   printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
result = 1235
```

Important

For parent, the return value of `fork()` is the PID of the created child.

PID 1234 (running)

PID 1235 (waiting)

在父进程中，fork()的返回值是子进程的pid.

```
1  int main(void) {
2    int result;
3    printf("before fork ...\n");
4    result = fork();
5    printf("result = %d.\n", result);
6
7    if(result == 0) {
8      printf("I'm the child.\n");
9      printf("My PID is %d\n", getpid());
10   }
11   else {
12     printf("I'm the parent.\n");
13     printf("My PID is %d\n", getpid());
14   }
15
16   printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
```

Important

For child, the return value of `fork()` is 0.

PID 1234 (stop)

PID 1235 (running)

在子进程中，fork()的返回值是0.

```
1  int main(void) {
2    int result;
3    printf("before fork ...\n");
4    result = fork();
5    printf("result = %d.\n", result);
6
7    if(result == 0) {
8      printf("I'm the child.\n");
9      printf("My PID is %d\n", getpid());
10   }
11   else {
12     printf("I'm the parent.\n");
13     printf("My PID is %d\n", getpid());
14   }
15
16   printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```

PID 1234 (stop)

PID 1235 (stop)

fork() System Call

- `fork()` behaves like "cell division".
  - It creates the child process by **cloning** from the parent process, including all user-space data, e.g.,

Cloned items	Descriptions
Program counter [CPU register]	That's why they both execute from the same line of code after `fork()` returns.
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file "fd", then the child will also have file "fd" opened automatically.

fork()会创建新的子进程，父进程的大部分值都会复制到该子进程中，只有少数值不同.

- `fork()` does not clone the following...

Distinct items	Parent	Child
Return value of `fork()`	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Parent.
Running time	Cumulated.	Just created, so should be 0.
[Advanced] File locks	Unchanged.	None.

exec() System Call family

- `execl()` - a member of the `exec` system call family (execl, execle, execlp, execv, execve, execvp).

```
int main(void) {
  printf("before execl ...\n");
  execl("/bin/ls", "/bin/ls", NULL);
  printf("after execl ...\n");
  return 0;
}
```

```
$ ./exec_example
before execl ...
```

Arguments of the `execl()` call

1st argument: the program name, "**`/bin/ls`**" in the example.
2nd argument: argument[0] to the program.
3rd argument: argument[1] to the program.

```
int main(void) {
  printf("before execl ...\n");
  execl("/bin/ls", "/bin/ls", NULL);
  printf("after execl ...\n");
  return 0;
}
```

```
$./exec_example
before execl ...
exec_example
exec_example.c
```

What is the output?

The same as **the output of running "ls" in the shell.**

exec()的一系列系统调用会启动新的进程，加载不同的程序

- Example #1: run the command `"/bin/ls"`

```
execl("/bin/ls", "/bin/ls", NULL);
```

Argument Order	Value in above example	Description
1	`"/bin/ls"`	The file that the programmer wants to execute.
2	`"/bin/ls"`	When the process switches to `"/bin/ls"`, this string is the **program argument[0]**.
3	`NULL`	This states the end of the program argument list.

- Example #2: run the command `"/bin/ls -l"`

```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

Argument Order	Value in above example	Description
1	`"/bin/ls"`	The file that the programmer wants to execute.
2	`"/bin/ls"`	When the process switches to `"/bin/ls"`, this string is the **program argument[0]**.
3	`"-l"`	When the process switches to `"/bin/ls"`, this string is the **program argument[1]**.
4	`NULL`	This states the end of the program argument list.

The shell prompt appears!

```
int main(void) {
  printf("before execl ...\n");
  execl("/bin/ls", "/bin/ls", NULL);
  printf("after execl ...\n");
  return 0;
}
```

```
$./exec_example
before execl ...
exec_example
exec_example.c
$ _
```

The output says:
(1) The gray code block **is not reached**!
(2) The process is **terminated**!

WHY IS THAT?!

exec()调用后不会返回原来的程序

```
/* code of program exec_example */
int main(void) {
  printf("before execl ...\n");
  execl("/bin/ls", "/bin/ls", NULL);
  printf("after execl ...\n");
  return 0;
}
```

Address Space of the process

```
/* Code of program "ls" */
int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```

Address Space of the process

exec() loads program "ls" into the memory of this process

```
/* Code of program "ls" */
int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```

Address Space of the process

The "**return**" or the "**exit()**" statement in "**/bin/ls**" will terminate the process...

Therefore, it is certain that the process cannot go back to the old program!

exec() 后进程会进入新程序，不再返回.

exec() Summary

- The process is changing the code that is executing and never returns to the original code.
  - The last two lines of codes are therefore not executed.
- The process that calls an exec* system call will replace user-space info, e.g.,
  - Program Code
  - Memory: local variables, global variables, and dynamically allocated memory;
  - Register value: e.g., the program counter;
- But, the kernel-space info of that process is preserved, including:
  - PID;
  - Process relationship;
  - etc.

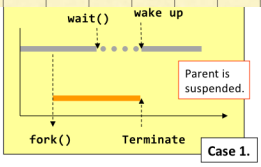
进程中的代码会改变且永不返回原代码.

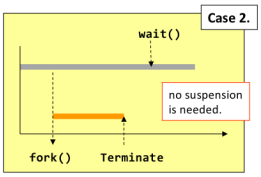
即会取代用户空间信息

进程的内核空间的信息会保留

wait()

- wait() suspends the calling process to waiting
- wait() returns when
  - one of its child processes changes from running to terminated.
- Return immediately (i.e., does nothing) if
  - It has no children
  - Or a child terminates before the parent calls wait for





挂起正在运行的进程

当某个子进程终止时返回

若①没有子进程 ②在wait()调用之前子进程已停止，则会立刻返回.

wait() v.s. waitpid()

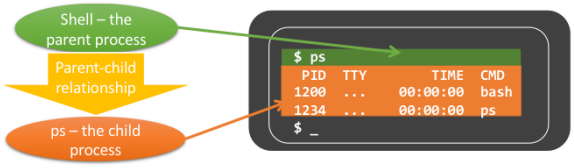
- wait()
  - Wait for any one of the child processes
  - Detect child termination only
- waitpid()
  - Depending on the parameters, waitpid() will wait for a particular child only
  - Depending on the parameters, waitpid() can detect different status changes of the child (resume/stop by a signal)

wait()会等待任一子进程，且仅检测子进程停止

waitpid()会等待有特定pid的进程，且检测该子进程所有的状态变化.

Implement Shell with fork(), exec(), and wait()

- A shell is a CLI
  - Bash in linux
  - invokes a function fork() to create a new process
  - Ask the the child process to exec() the target program
  - Use wait() to wait until the child process terminates

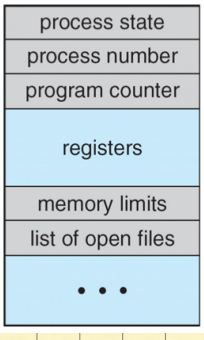


Processes: Kernel View

Process Control Block (PCB) 进程控制块

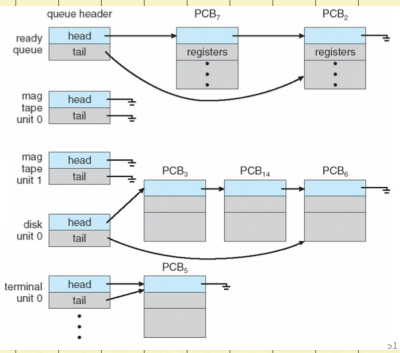
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



Ready Queue And I/O Device Queues

- PCBs are linked in multiple queues
  - Ready queue contains all processes in the ready state (to run on this CPU)
  - Device queue contains processes waiting for I/O events from this device
  - Process may migrate among these queues



PCB连接在多个队列中.

Ready Queue: 在ready态的进程

Device Queue: 等待IO的进程

进程可能在多个队列中迁移

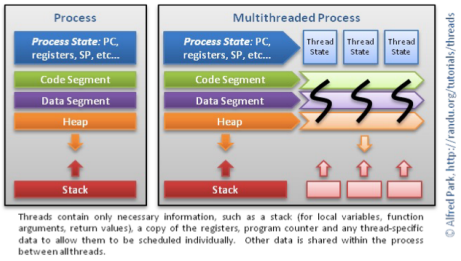
Threads 线程

- One process may have more than one threads
  - A single-threaded process performs a single thread of execution
  - A multi-threaded process performs multiple threads of execution "concurrently", thus allowing short response time to user's input even when the main thread is busy
- PCB is extended to include information about each thread

一个进程中可以有不止一个线程

PCB会扩展，包括每个线程的信息

- Single threaded process and multi-threaded process



Switching Between Processes

- Once a process runs on a CPU, it only gives back the control of a CPU
  - when it makes a system call
  - when it raises an exception
  - when an interrupt occurs
- What if none of these would happen for a long time?
  - Coorperative scheduling: OS will have to wait
    - Early Macintosh OS, old Alto system
  - Non-coorperative scheduling: timer interrupts
    - Modern operating systems
- When OS kernel regains the control of CPU
  - It first completes the task
    - Serve system call, or
    - Handle interrupt/exception
  - It then decides which process to run next
    - by asking its **CPU scheduler**
    - How does it make decisions?
    - More about CPU scheduler later
  - It performs a **context switch** if the soon-to-be-executing process is different from the previous one

只有当进入内核态时，CPU才从进程移交给OS
(system call、exception、interrupt)
若长时间不进入内核态：
①协作型调度：OS会一直等待
②非协作型调度：计时中断

CPU调度器会决定下一个进程

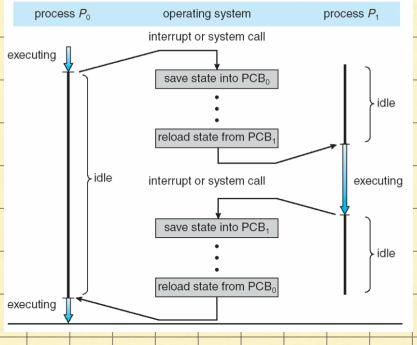
当下一个进程与当前进程不一样时，会发生上下文切换.

Context Switch

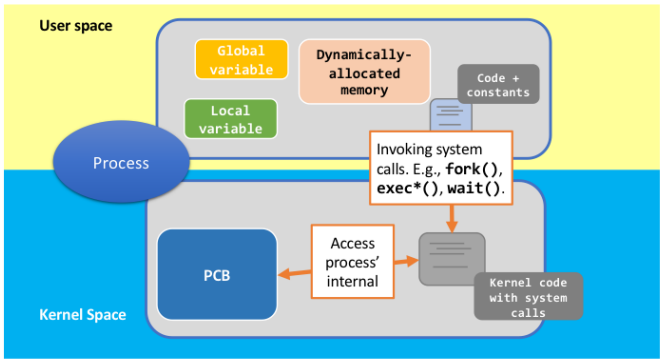
- During context switch, the system must save the state of the old process and load the saved state for the new process
- Context of a process is represented in the PCB
- The time used to do context switch is an overhead of the system; the system does no useful work while switching
  - Time of context switch depends on hardware support
  - Context switch cannot be too frequent

在上下文切换时，系统要保存旧进程的状态并加载新进程的状态
进程的上下文信息在PCB里.

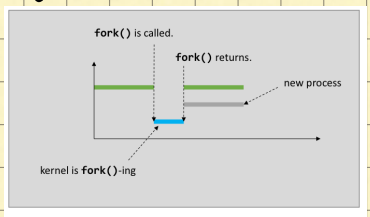
上下文切换用时看硬件，不要过于频繁



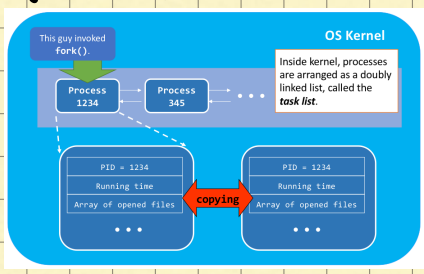
Recall: fork(), exec(), and wait()

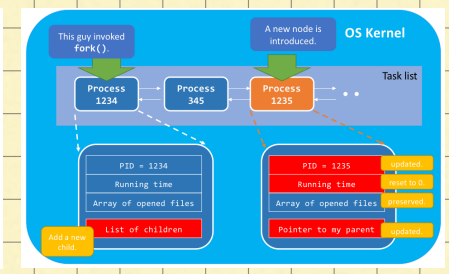


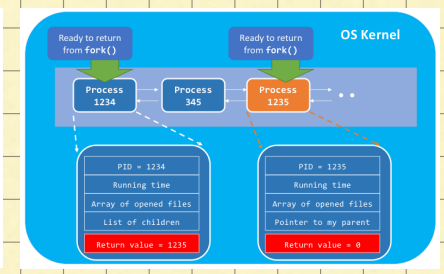
fork() in User Mode



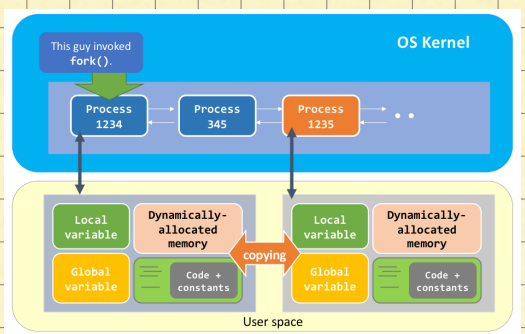
fork(): Kernel View





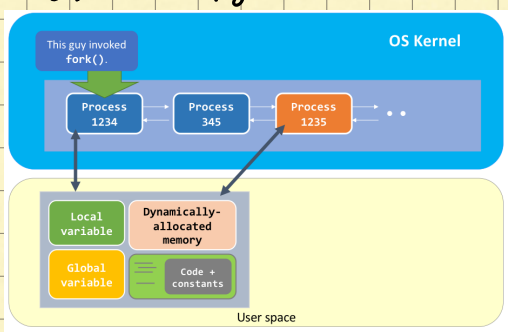


Case 1: Duplicate Address Space



直接完全复制地址空间

Case 2: Copy on Write



当子进程修改时，才会复制修改的部分

两种实现都有，希望父子进程的地址空间是不一样的。

fork(): Opened Files

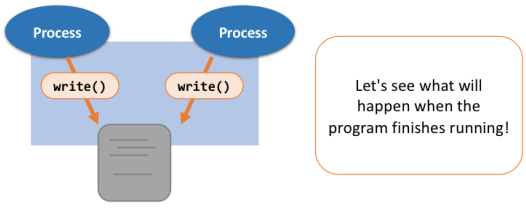
父子进程共享文件状态.

- Array of opened files contains:

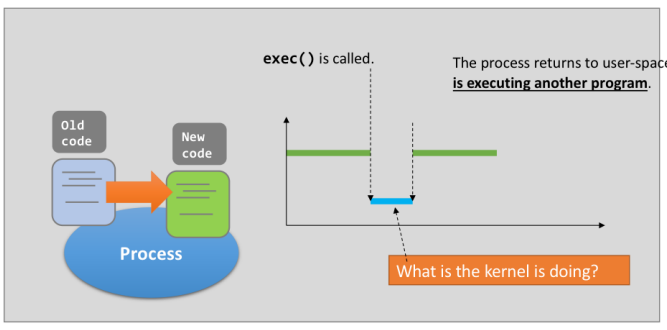
Array Index	Description
0	Standard Input Stream; **FILE *stdin;**
1	Standard Output Stream; **FILE *stdout;**
2	Standard Error Stream; **FILE *stderr;**
3 or beyond	Storing the files you opened, e.g., **fopen()**, **open()**, etc.

- That's why a parent process shares the same terminal output stream as the child process.

- What if two processes, sharing the same opened file, write to that file together?

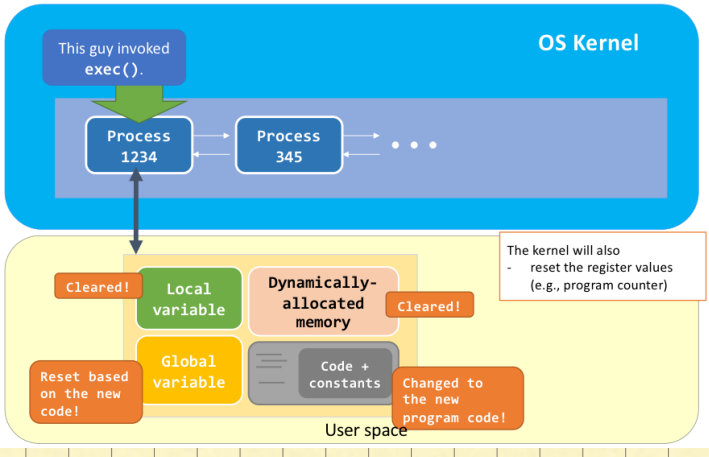


exec() in User Mode



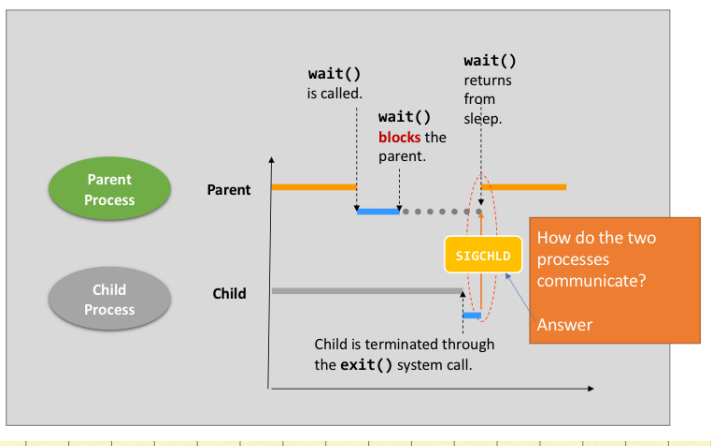
返回user space后执行的是另一个程序

exec(): Kernel View

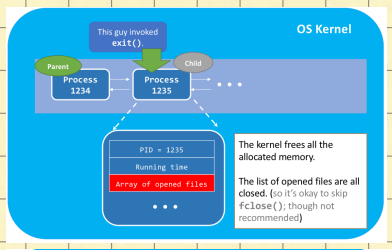


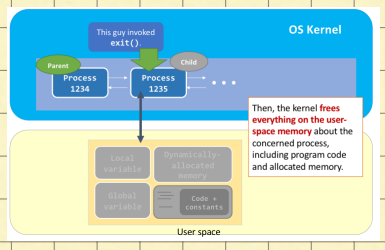
Local variable与dynamically allocated都被清除,
Global与code+constant都变成新程序的

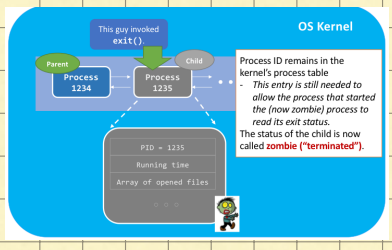
wait() and exit()

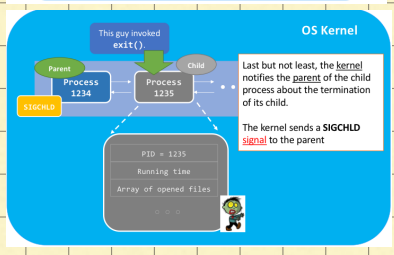


exit() Kernel View



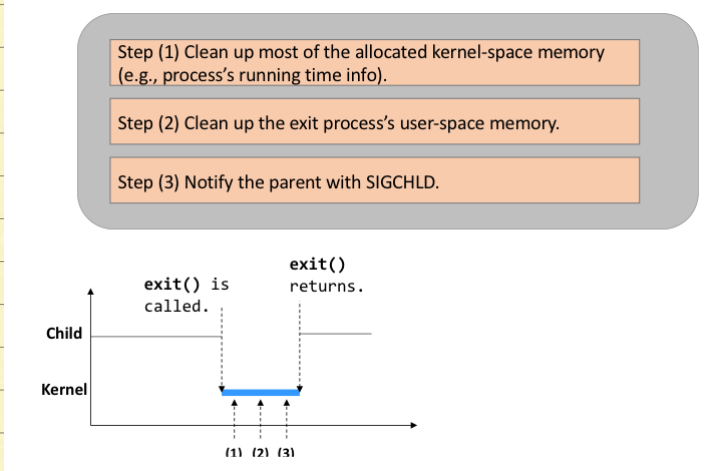




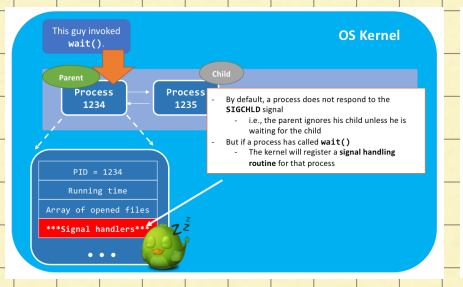


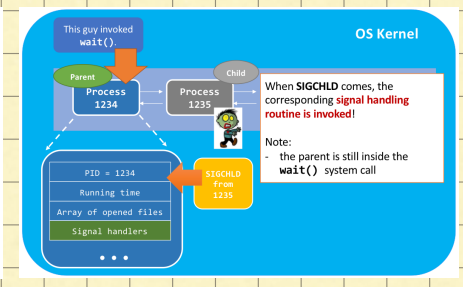
① 释放内存
② 进入僵尸状态
③ 向父进程发送SIGCHLD信号

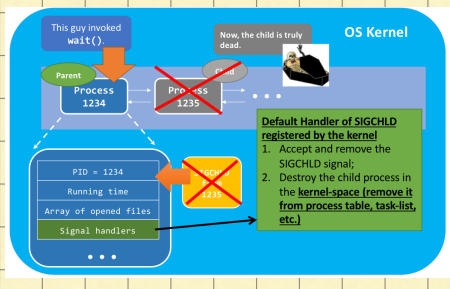
exit() Summary

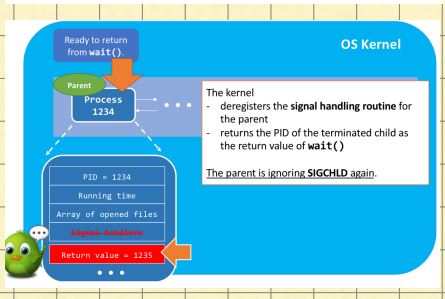


wait() Kernel View









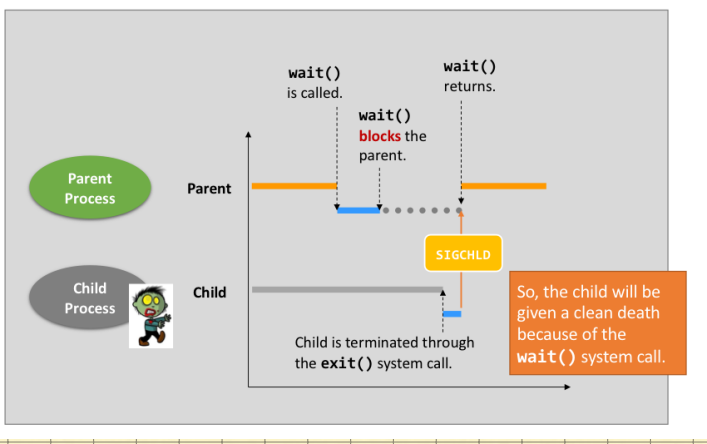
① 一般情况下，进程会忽略 SIGCHLD 信号
但若进程调用过 wait()，kernel会为该进程注册 signal handling routine.

② 当 SIGCHLD 到来时，对应的 signal handling routine 会被调用

③ signal handler 会移除 SIGCHLD 信号，并将 kernel space 的子进程销毁.

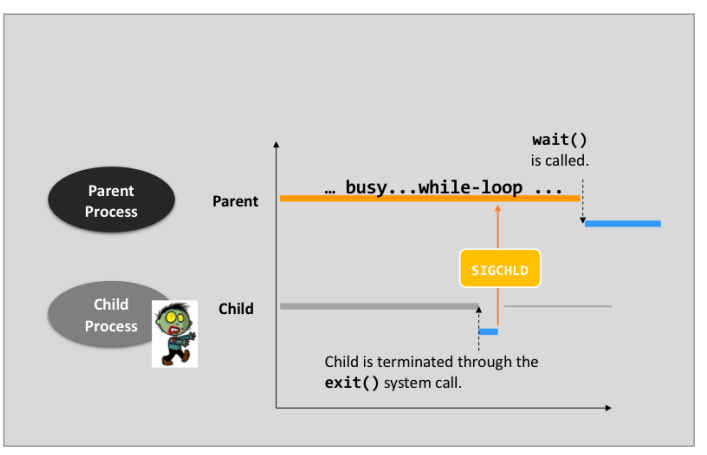
④ 父进程的 signal handler 会删除，并返回子进程 pid.

Normal Case

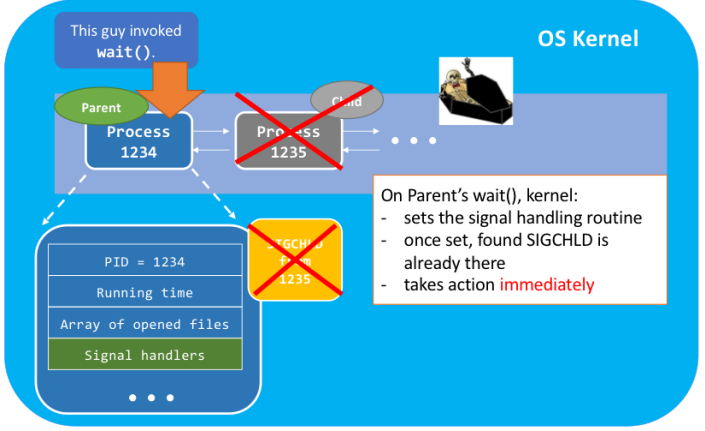


当父进程收到 SIGCHLD 时会执行相关操作后结束 wait().

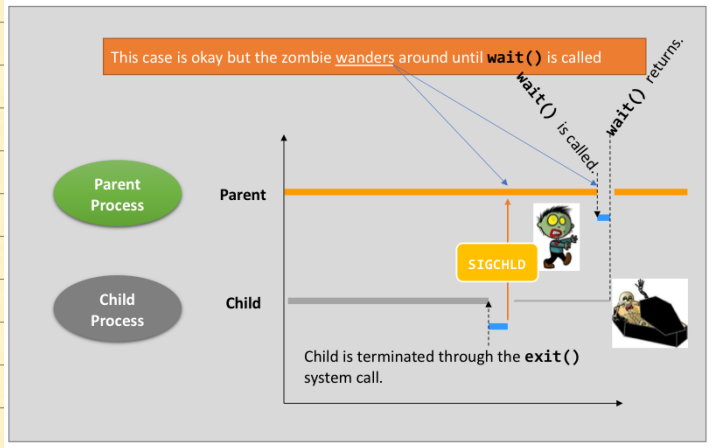
Parent's wait() after Child's exit()



若子进程死亡时父进程仍在执行，SIGCHLD 会在父进程中排队.



当父进程进入wait()时，若发现SIGCHLD队列中有内容，会立刻处理.



Summary of wait() & exit()

- **exit()** system call turns a process into a zombie when...
  - The process calls **exit()**.
  - The process returns from **main()**.
  - The process terminates abnormally.
    - The kernel knows that the process is terminated abnormally. Hence, the kernel invokes **exit()** for it.
- **wait()** & **waitpid()** reap zombie child processes.
  - It is a must that you should never leave any zombies in the system.
  - **wait()** & **waitpid()** pause the caller until
    - A child terminates/stops, OR
    - The caller receives a signal (i.e., the signal interrupted the wait())
- Linux will label zombie processes as "**<defunct>**".
  - To look for them:

```
$ ps aux | grep defunct
.....  3150 ... [ls] <defunct>
$ _
```

PID of the process

```
1  int main(void)
2  {
3      int pid;
4      if( (pid = fork()) !=0 ) {
5          printf("Look at the status of the child process %d\n", pid);
6          while( getchar() != '\n' );
7          wait(NULL);
8          printf("Look again!\n");
9          while( getchar() != '\n' );
10     }
11     return 0;
12 }
```

"enter" here

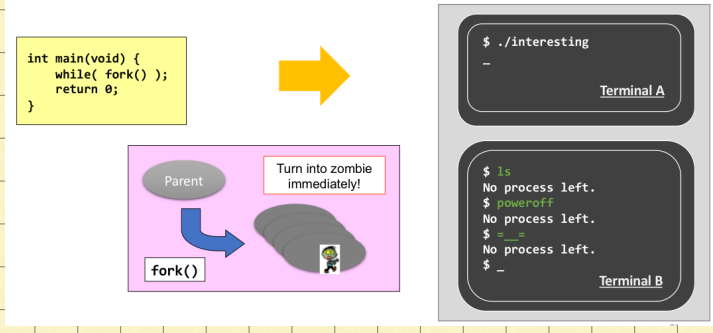
"enter" here

This program requires you to type "enter" twice before the process terminates.

You are expected to see **the status of the child process changes (ps aux [PID])** between the 1st and the 2nd "enter".

Using wait() for Resource Management

- It is not only about process execution / suspension...
- It is about system resource management.
  - A zombie takes up a PID;
  - The total number of PIDs are limited;
    - Read the limit: "cat /proc/sys/kernel/pid_max"
    - It is 32,768.
  - What will happen if we don't clean up the zombies?



The first process

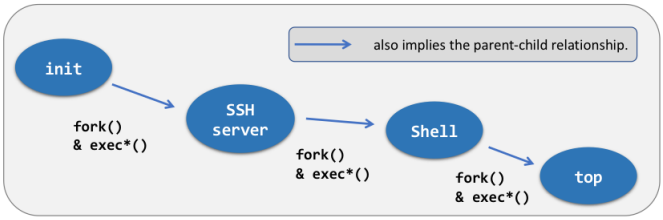
- We now focus on the process-related events.
  - The kernel, while it is booting up, creates the first process – init.
- The "init" process:
  - has PID = 1, and
  - is running the program code "/sbin/init".
- Its first task is to create more processes...
  - Using fork() and exec().

How does uCore create the first process?

在内核启动时会创建第一个进程init，该进程是所有进程的祖先。

A Tree of Processes

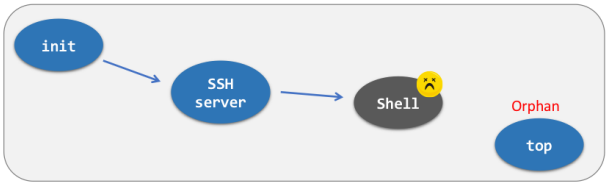
- You can view the tree with the command:
  - "**pstree**"; or
  - "**pstree -A**" for ASCII-character-only display.



Orphans

- However, termination can happen, at any time and in any place...
  - This is no good because an orphan turns the hierarchy from a **tree** into a **forest**!
  - Plus, no one would know the termination of the orphan.

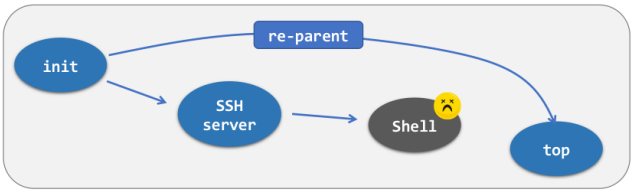
当某进程的父进程先结束了，该进程即为孤儿进程。



Re-parent

- In Linux
  - The "**init**" process will become the step-mother of all orphans
  - It's called **re-parenting**
- In Windows
  - It maintains a *forest-like process hierarchy*.....

当出现孤儿进程后，该进程的父进程会设成init。

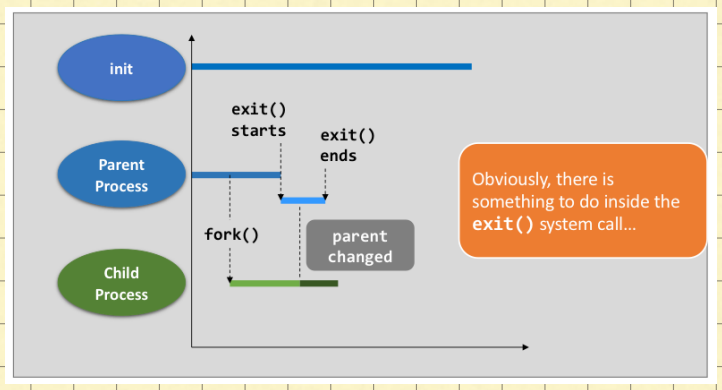


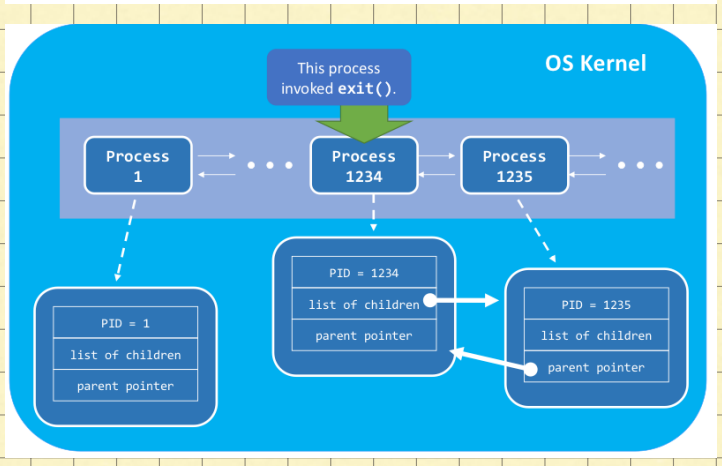
Example

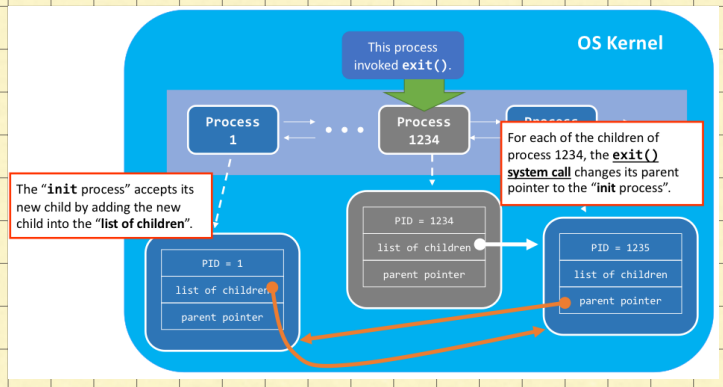
```
1  int main(void) {
2      int i;
3      if(fork() == 0) {
4          for(i = 0; i < 5; i++) {
5              printf("(%d) parent's PID = %d\n",
6                      getpid(), getppid() );
7          sleep(1);
8          }
9      }
10     else
11         sleep(1);
12     printf("(%d) bye.\n", getpid());
13 }
```

```
$ ./reparent
(1235) parent's PID = 1234
(1235) parent's PID = 1234
(1234) bye.
$ (1235) parent's PID = 1
(1235) parent's PID = 1
(1235) parent's PID = 1
(1235) bye.
$ _
```

getppid() is the system call that returns the parent's PID of the calling process.

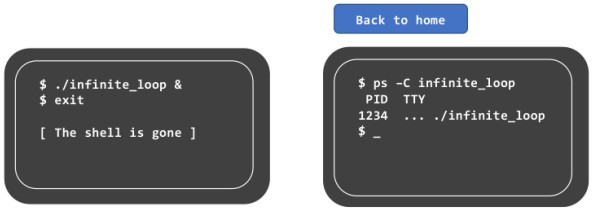




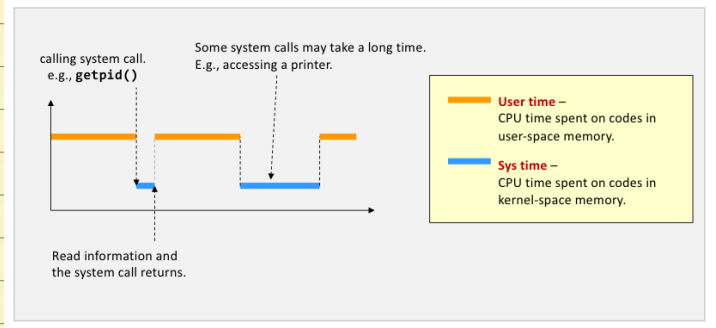


Background Jobs

- The re-parenting operation enables something called **background jobs** in Linux
  - It allows a process runs **without a parent terminal/shell**



Measure Process Time

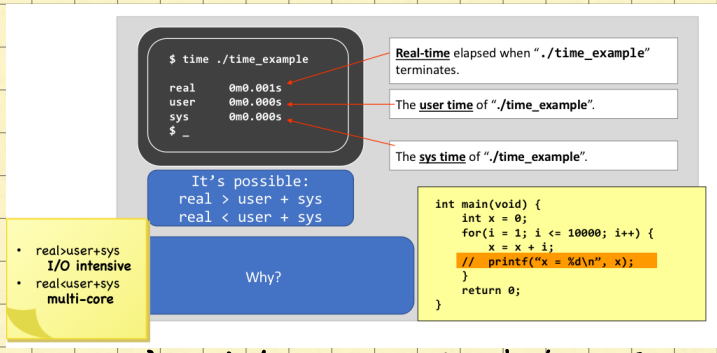


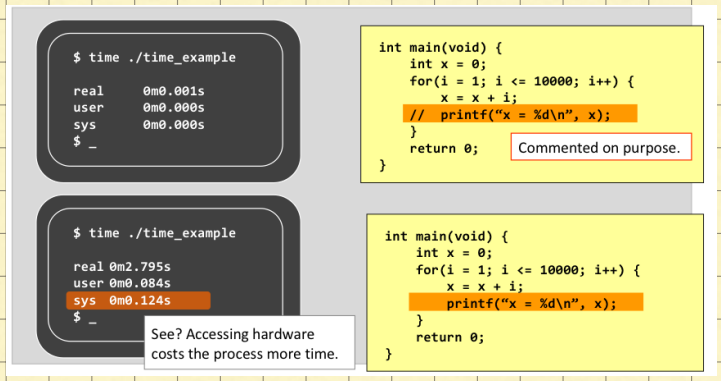
用户时间：花在用户空间中的CPU时间

系统时间：花在内核空间中的CPU时间

一般来说程序用时是这两个时间相加.

User Time v.s. Sys Time





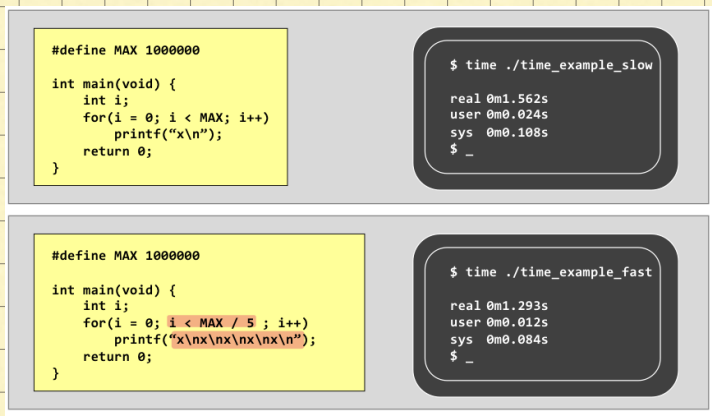
I/O密集会经常用户/内核模式切换

多核会统计每个核中的总用时.

- The user time and the sys time together **define the performance of an application.**
  - When writing a program, you must consider both the user time and the sys time.
    - E.g., the output of the following two programs are exactly the same. But, their running time is not.

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX; i++)
        printf("x\n");
    return 0;
}
```

```
#define MAX 1000000

int main(void) {
    int i;
    for(i = 0; i < MAX / 5 ; i++)
        printf("x\nx\nx\nx\nx\n");
    return 0;
}
```



printf()是API，里面有buffer，会在buffer积累到一定程度后才系统调用.

Thead

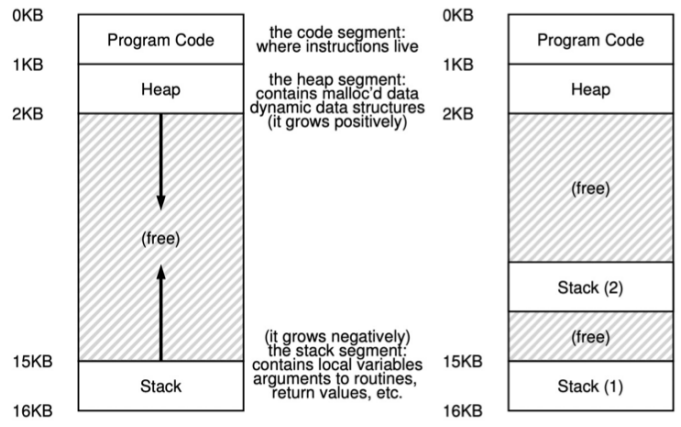
- Thread is an **abstraction** of the execution of a program
  - A single-threaded program has one point of execution
  - A multi-threaded program has more than one points of execution
- Each thread has its own **private** execution state
  - Program counter and a private set of registers
  - A private stack for thread-local storage
  - CPU switching from one thread to another requires context switch
- Threads in the same process **share** computing resources
  - Address space, files, signals, etc.

线程是程序执行的抽象.

线程有私有执行空间

同一进程中的线程共享计算资源.

Single-Threaded & Multi-Threaded



Thread Pros.

- Increase parallelism
  - One thread per CPU makes better use of multiple CPUs to improve efficiency
- Avoid blocking program progress due to slow I/O
  - Threading enables overlap of I/O with other activities within a single program
  - e.g., many modern server-based applications (web servers, database management systems, and the like) make use of threads
- And allow resource sharing !!!

提高并行

避免因IO的阻塞

资源共享

Thread Implementation

- User-level thread
  - Thread management (e.g., creating, scheduling, termination) done by user-level threads library
  - OS does not know about user-level thread
- Kernel-level thread
  - Threat management done by kernel
  - OS is aware of each kernel-level thread

用户级线程:

OS并不知道user-level thread

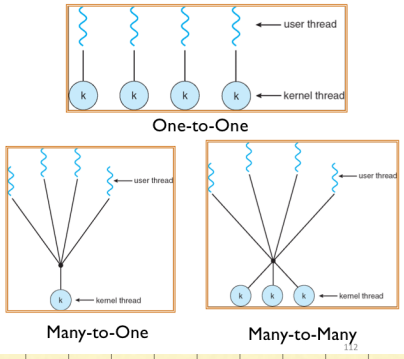
内核级线程:

OS会关心每一个kernel-level thread

Thread Models

Thread Models

- One-to-one mapping
  - One user-level thread to one kernel-level thread
- Many-to-one mapping
  - Many user-level thread to one kernel-level thread
- Many-to-many mapping
  - Many user-level thread to many kernel-level thread



Map: 将用户级线程映射到内核级线程.

Pros & Cons

- Many-to-one mapping
  - Pros: context switch between threads is cheap
  - Cons: When one thread blocks on I/O, all threads block
- One-to-one mapping
  - Pros: Every thread can run or block independently
  - Cons: Need to make a crossing into kernel mode to schedule
- Many-to-many mapping
  - Many user-level threads multiplexed on less or equal number of kernel-level threads
  - Pros: best of the two worlds, more flexible
  - Cons: difficult to implement