Оперативни Системи

Процеси

Проф. Д-р Димитар Трајанов
Вон. проф. Д-р Невена Ацковска
Доц. Д-р Боро Јакимовски

Процеси

- **Процеси се програми во извршување**
- Основна апстракција на ОС!
  - Овозможуваат системи со еден CPU да му делуваат на корисникот како системи со повеќе (виртуелни) CPU

Повеќе процеси во временски период

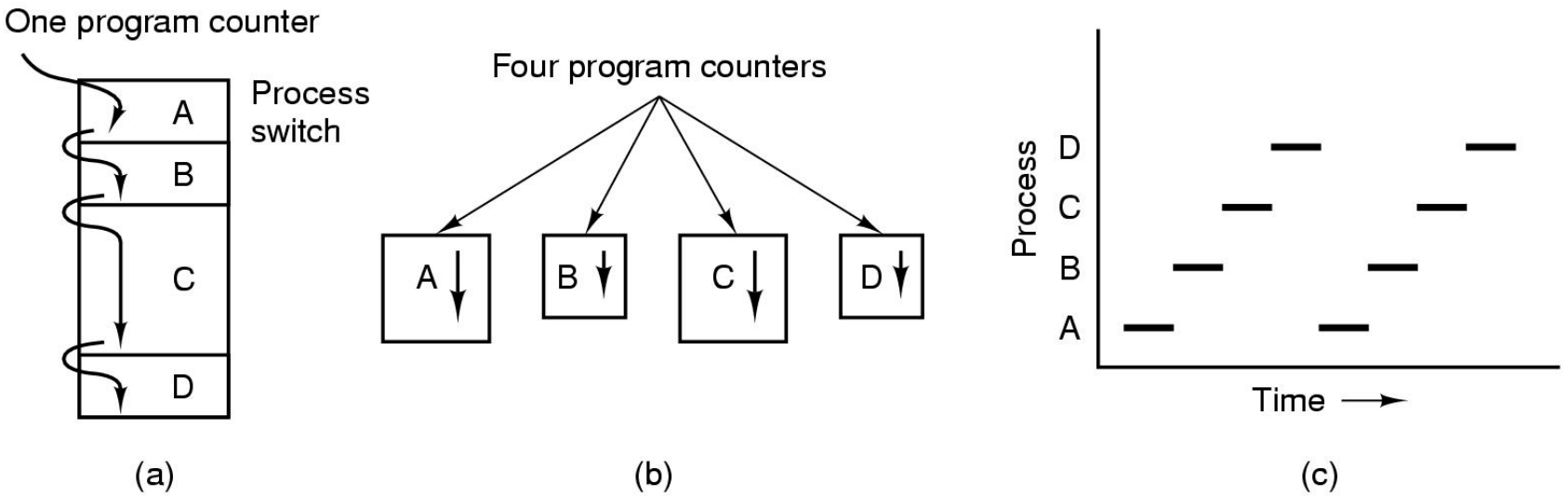
- Во секој миг паралелно работат повеќе процеси
  - ОС стартувани процеси
  - Кориснички стартувани процеси

- Корисникот има илузија дека се извршуваат паралелно

Псевдопаралелизам

- Илузија на паралелно извршување на повеќе процеси на еден CPU
  - На секој од процесите му се дава одредено време на CPU
- Креаторите на ОС направиле концептуален **модел на секвенционални процеси**

Процесен модел



a) Мултипрограмирање за четири процеси
b) Концептуален модел за 4 независни секвенцијални процеси
c) Само еден процес е активен во даден момент

A. S. Tanenbaum, Modern Operating Systems, 3rd Edition, Pearson Prentice Hall, 2009

Креирање на процес

Настани кои доведуваат до креирање на процеси

1. Иницијализација на системот
2. Креирање на процес од некој друг процес
3. Креирање на процес по барање на корисник
4. Иницирање на (batch) пакетни задачи

Запирање на процес

Услови за запирање на процес

1. Нормален излез (своеволно)
2. Излез поради грешка (своеволно)
3. Фатална грешка (насилно)
4. “Убивање” од друг процес (насилно)

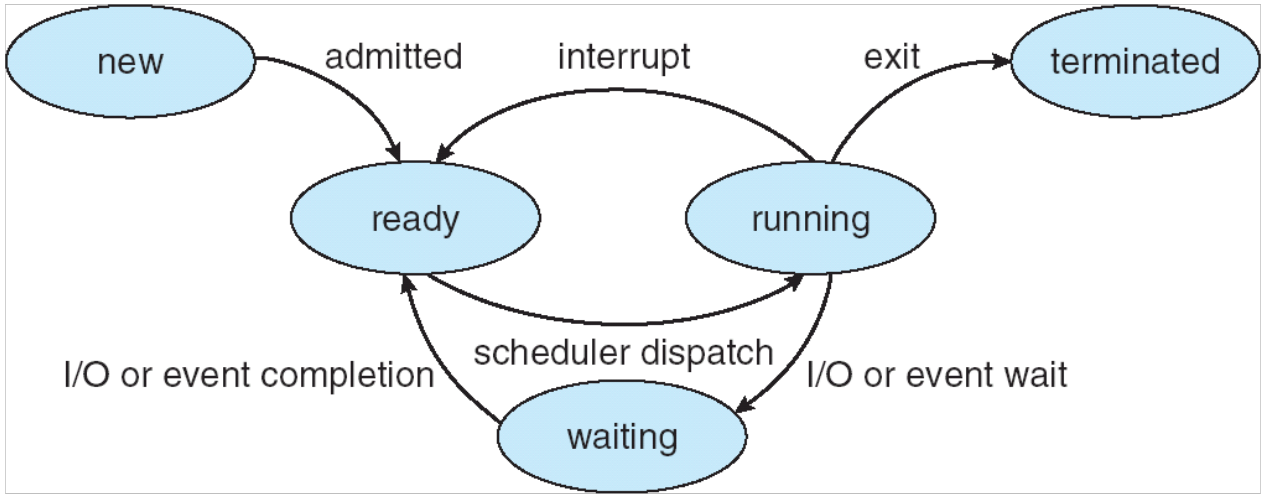
Хиерархија на процеси

- Родителите креираат деца процеси, децата може да креираат сови процеси
- Во UNIX се формира хиерархија и таа се нарекува група од процеси
- Windows нема концепт на хиерархиско формирање. Сите процеси се еднакви

Состојби на процесите

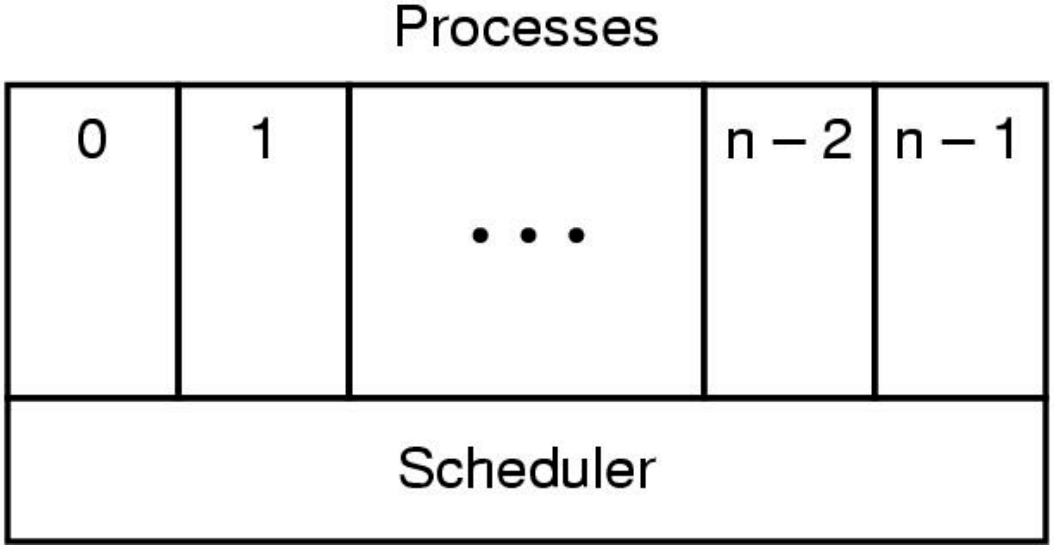
- Ги дефинира моменталната активност на процесот
  - **Нов**: процесот е креиран;
  - **Активен**: инструкциите се извршуваат;
  - **Чека (блокиран)**: процесот чека да се случи нешто (В/И или прием на сигнал);
  - **Спремен:** процесот чека да му биде назначен CPU – останатите ресурси му се доделени;
  - **Терминиран**: процесот завршил;

Животен циклус на процес



- Транзиции:
  - Програмски акции (системски повици)
  - ОС акции (распределување)
  - Надворешни настани (прекини, interrupts)

Процесен модел во оперативен систем



- Најниското ниво ги обработува прекините и врши распределба на извршувањето на процесите – распоредувач
- Повисокото ниво се процеси за кои може да се смета дека се извршуваат секвенцијално

Имплементација на процесен модел

- ОС одржува табела на процеси
  - Еден влез по процес
  - Претставува низа на структури

- За секој процес постои *контролен блок на процесот* (Process Control Block – PCB) кој
  - ги опишува неговите компоненти
  - дозволува ***ефикасен*** и ***централизиран*** пристап до сите информации во врска со процесот

  - Редовите на чекање обично користат покажувач кон PCB-ата

Информации во PCB

- **За секој процес:**

- **идентификационен број на процесот (PID)**
  - еднозначен
- **Состојба на процесот**
- **Програмски бројач**
  - адреса на следната инструкција што треба да се изврши за тој процес;
- **Регистри на CPU**
  - зачувување на информацијата за состојбата при прекин, за да може процесот да продолжи каде што застанал;
- **Информација за CPU** – распоредување
  - Покажувачи за редовите на чекање приоритет на процеси;

Информации во PCB (2)

- **Информации за управување со меморија**
  - мемориски описи (во зависност од тоа каква меморија користи ОС);
- **Кориснички идентификациони броеви**
  - uid, gid, euid, egid
- **Статус на В/И**
  - Информацијата вклучува листа на В/И уреди доделени на тој процес, листа на отворени датотеки...
- **Информации за:**
  - Време на користење на CPU;
  - реално време на извршување
  - лимити и квоти на користење
- **Покажувачи кон таткото, децата**

PCB елементи

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Полиња во табелата за процеси

Контекст на процес

- **Контекстот** е претставен **во PCB** и се состои од:
  - Вредност на CPU регистри;
  - Статус (состојба) на процес;
  - Информација за управување со меморија

- Процесот има свое множество “приватни” CPU регистри
  - во главната меморија
  - се полнат со информација кога процесот преминува од спремен во активен
  - мора да се зачуваат назад во главната меморија кога процесот преминува од “активен” во спремен или чека В/И

Промена на контекст

- ***Промена на контекстот*** *(context switch)* – доделување, промена на CPU на друг процес, се прави кога се случува прекин, системски повик или според режим на работа;

- Тоа е скапа операција
  - Кернелот го снима контекстот на стариот процес во неговиот PCB
  - Кернелот го вчитува контекстот на новиот процес кој треба да се извршува;
  - Таа е битен фактор на ефикасноста на ОС и нејзината цена продолжува да расте со забрзувањето на CPU
  - Покомплексен ОС – повеќе работа при промена на контекстот

Кога се менуваат процеси?

- Прекин од истечено време (clock interrupt)
  - процесот го потрошил доделеното време
- I/O прекин
- Мемориска грешка (Memory fault)
  - мемориската адреса е во виртуелната меморија, па мора прво да се доведе до работната
- Замка (Trap)
  - се случила грешка
  - може да доведе процесите да преминат во Exit состојба
- Повик
  - како отворање на датотека

UNIX-создавање, извршување и завршување процеси

- Нов процес се креира со fork()
- Изведување на програма – exec() фамилија
- Завршување – exit()

UNIX-овиот fork() (1)

- Креира дете – процес
  - два различни процеса извршуваат копија од еден ист програм
- Детето – процес наследува од таткото:
  - идентични копии на променливите и меморијата (адресен простор)
  - идентични копии на сите CPU регистри (освен еден)

UNIX-овиот fork() (2)

- Двата процеса (таткото и детето) се извршуваат од истата точка по враќањето од повикот fork():
  - за детето – процес fork() враќа 0 (PID за детето)
  - за таткото – процес, fork() го враќа идентификациониот број на процесот – дете
- Едноставната имплементација на fork():
  - алоцира меморија за детето – процес
  - ја копира меморијата и CPU регистрите од таткото во детето – процес
  - ***скапо!***

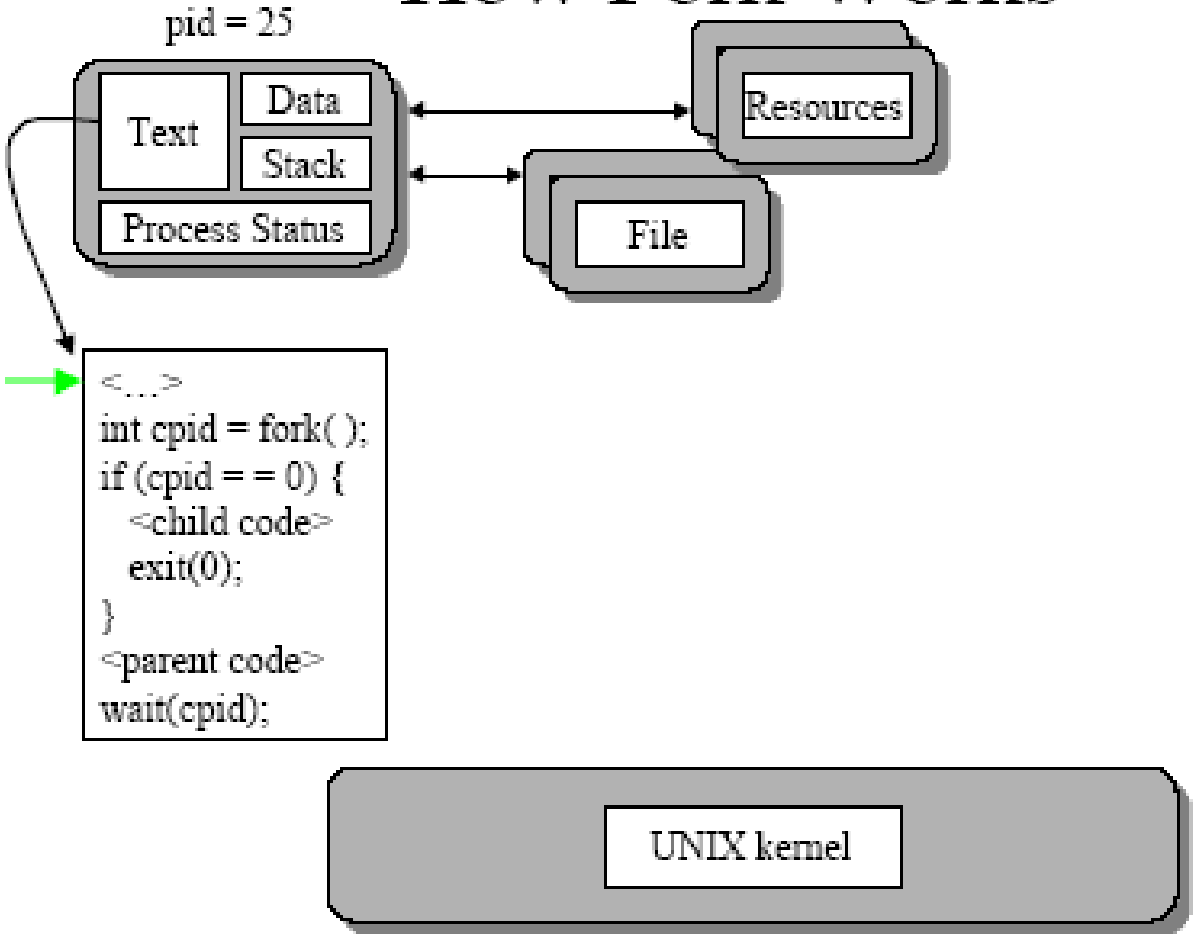
Користење на fork()

```
if (fork() == 0) {
…
}
…
```

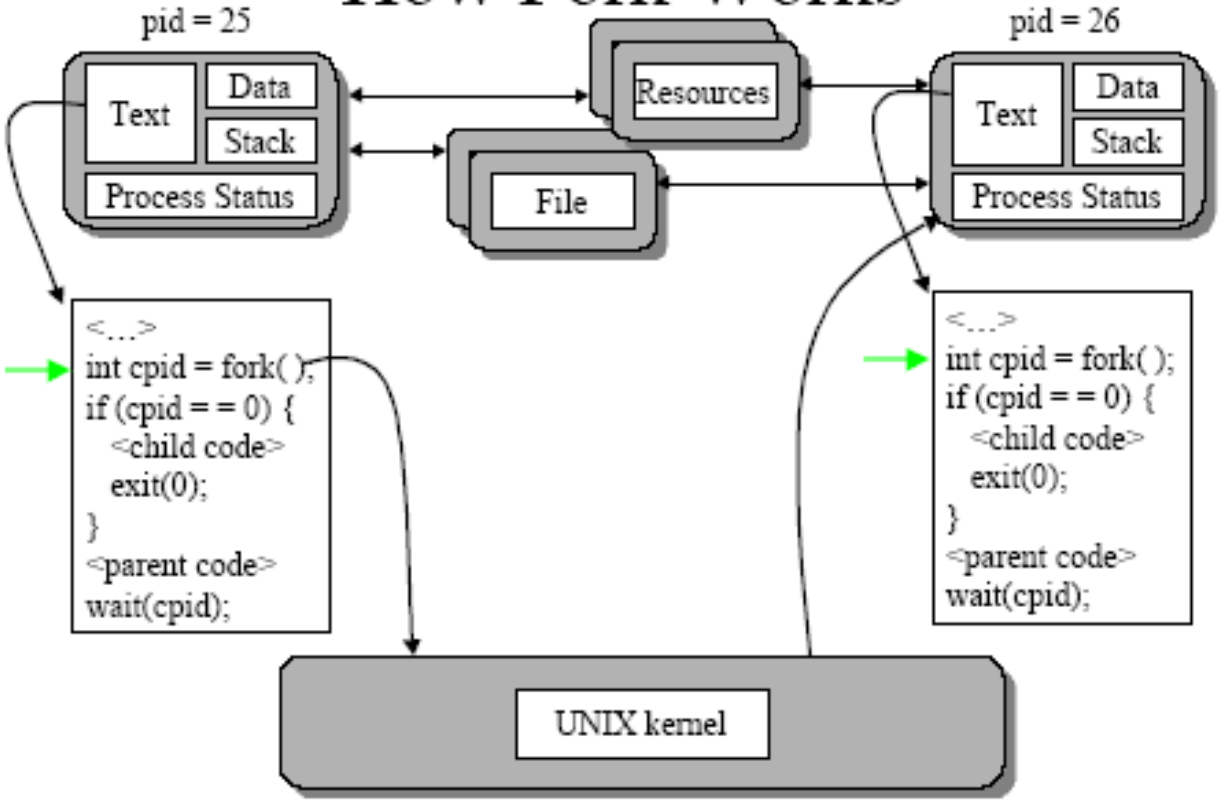
```
main()
…
int pid = fork();          // create a child

if (pid == 0) {            // child continues here
…
}
else {                     // parent continues here
…
}
```

- Кодот на таткото и на детето се во иста „програма“
- Детето ги наследува сите отворени датотеки и мрежни конекции

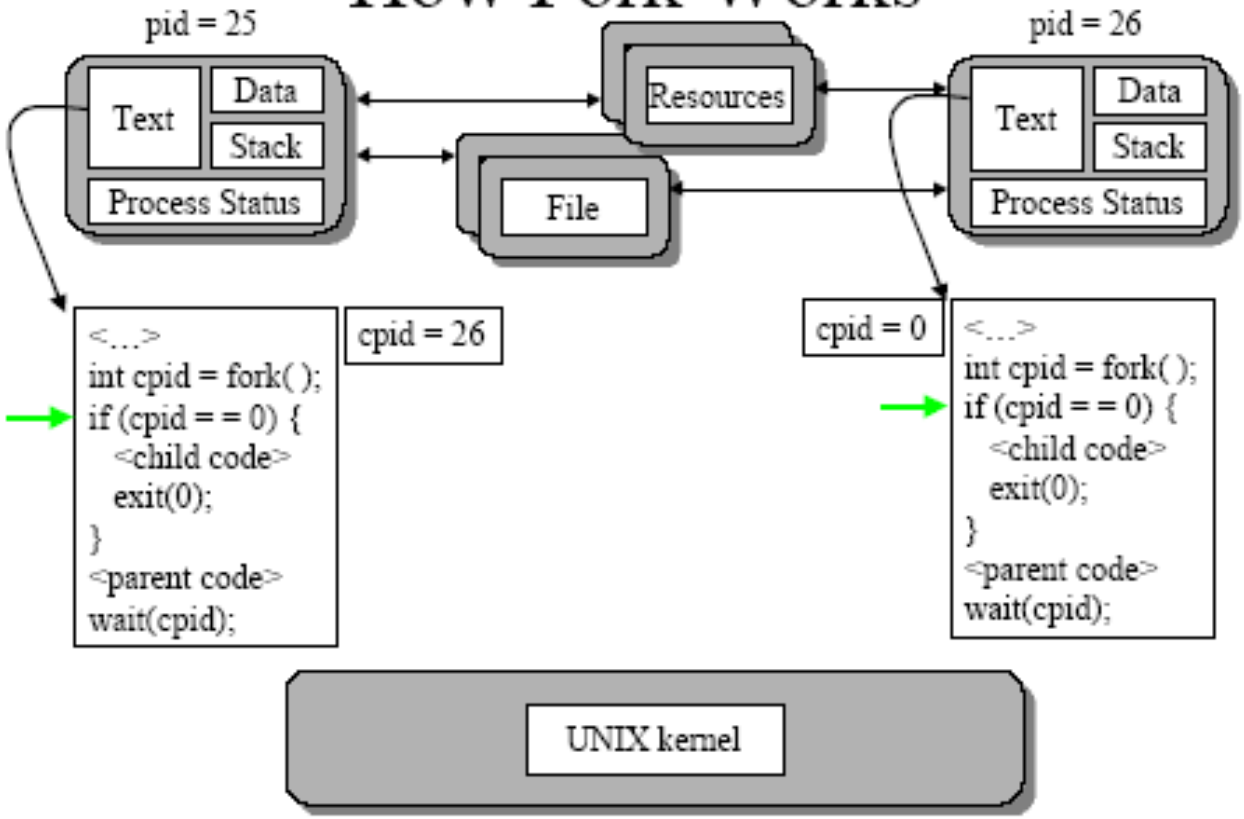
How Fork Works



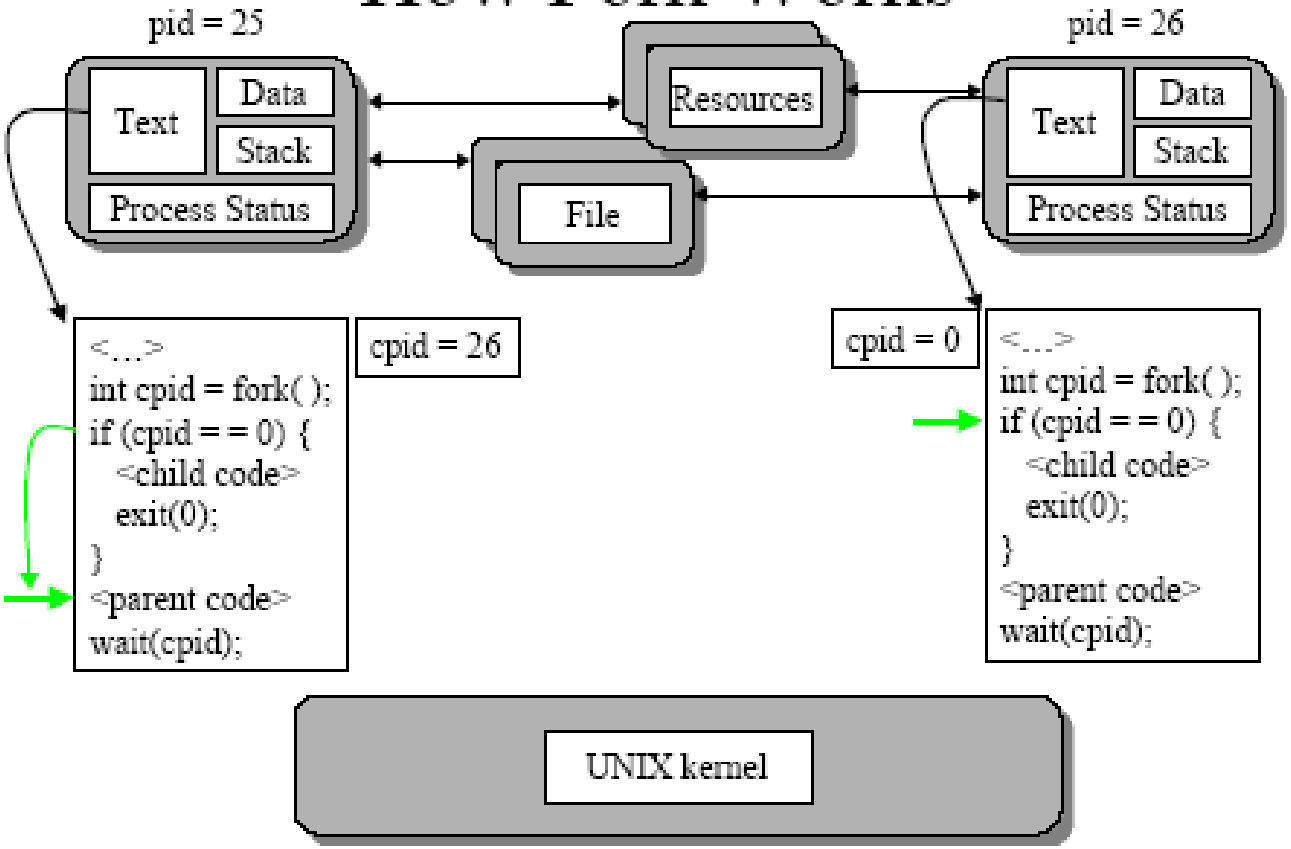
How Fork Works



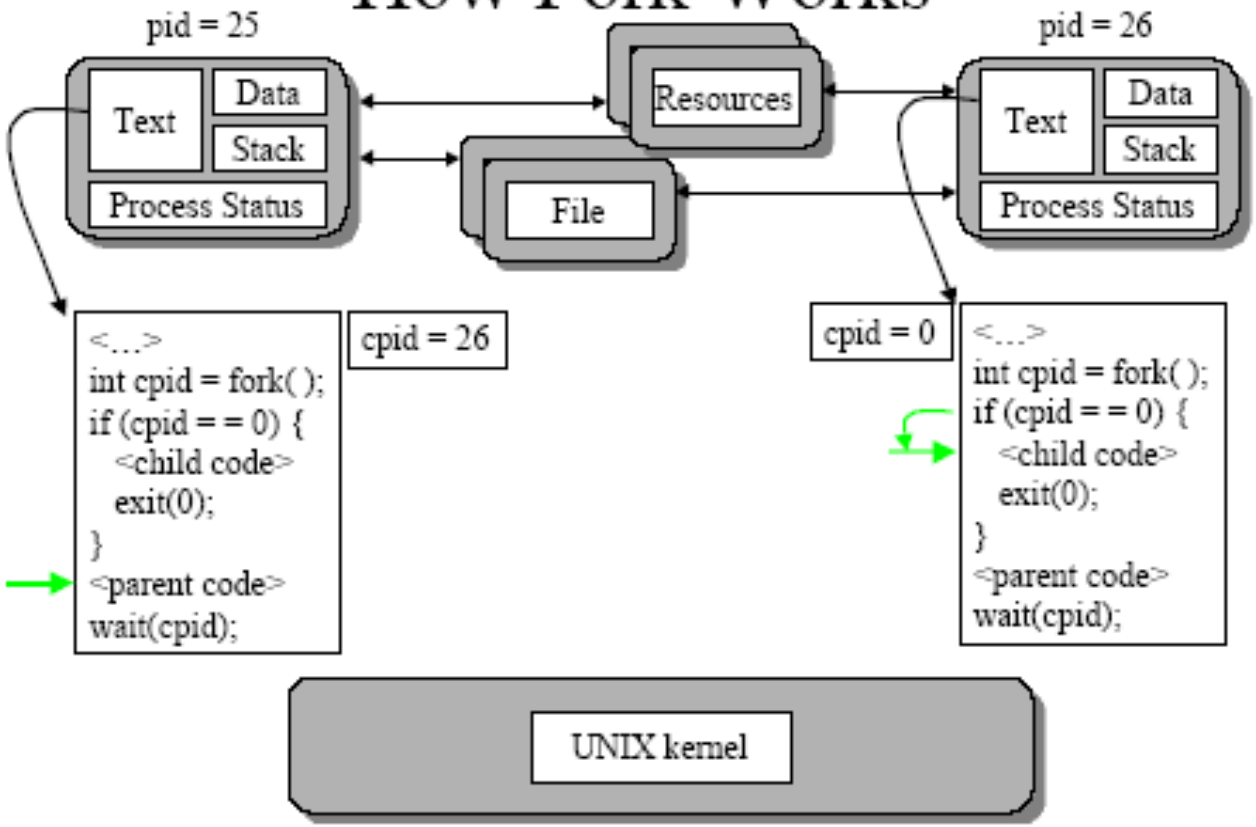
How Fork Works



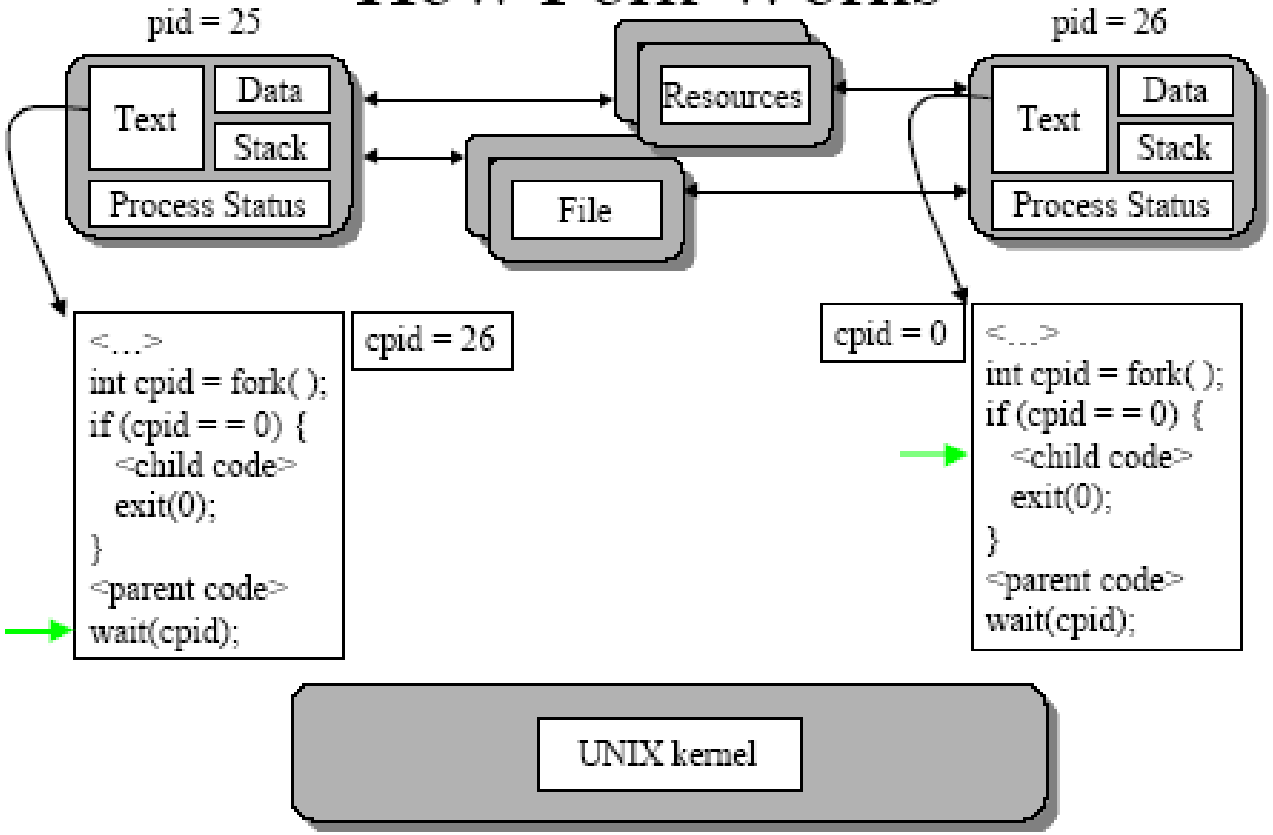
How Fork Works



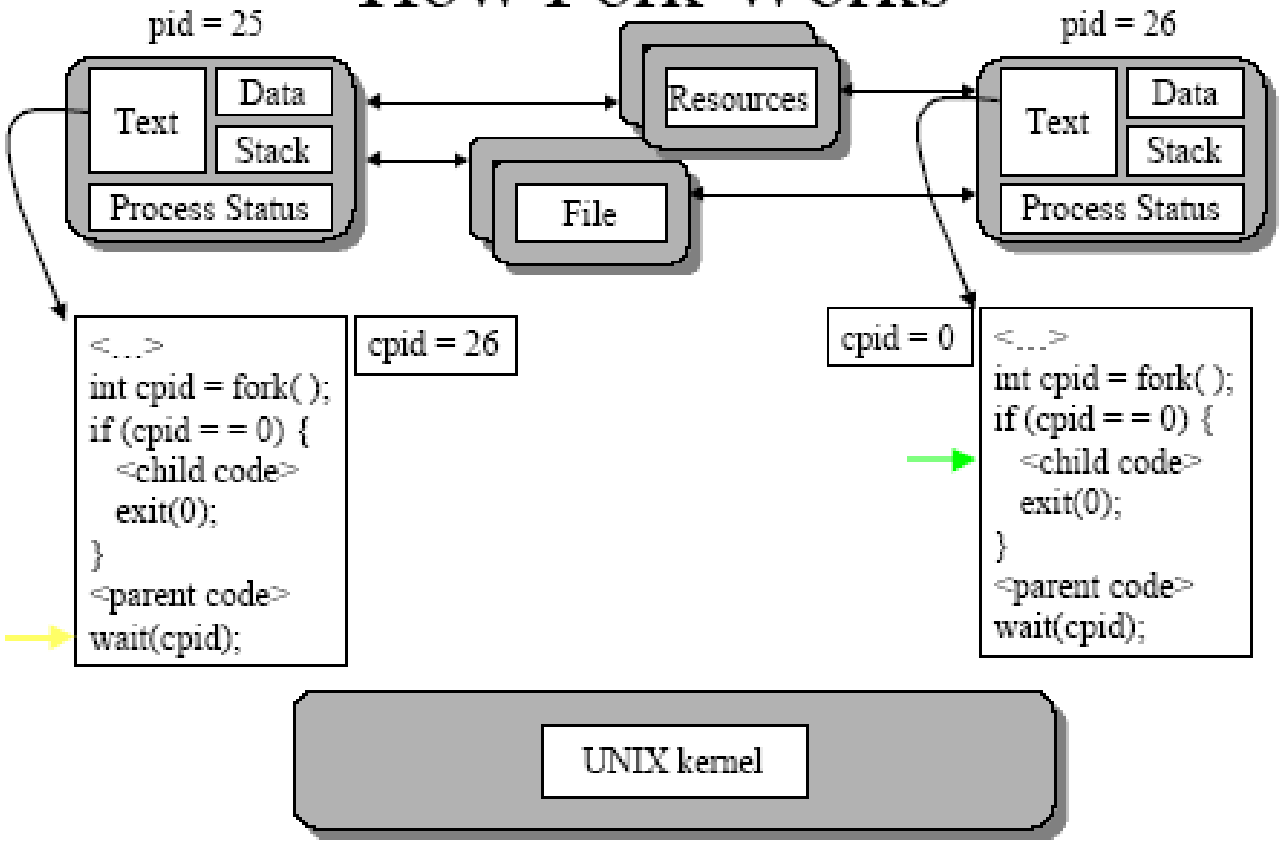
How Fork Works



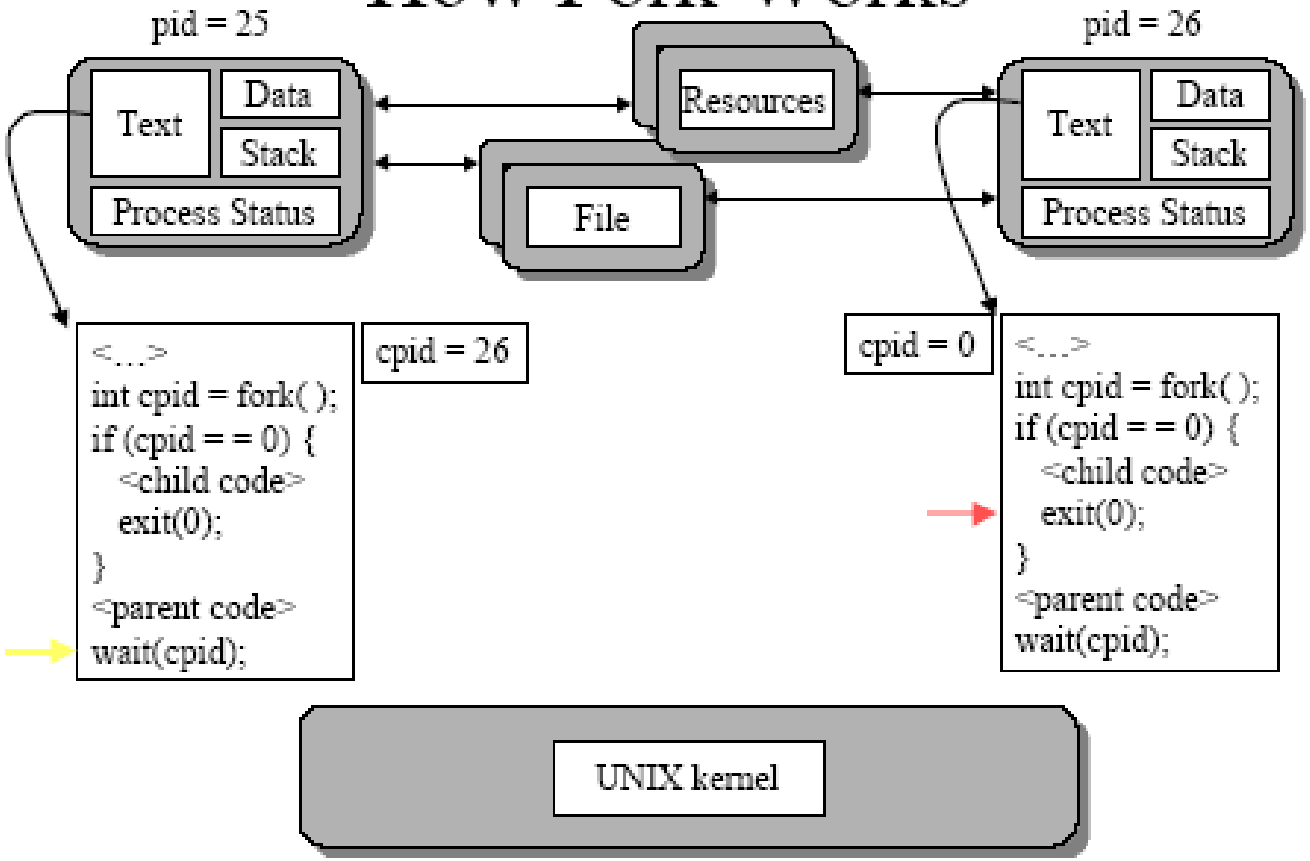
How Fork Works



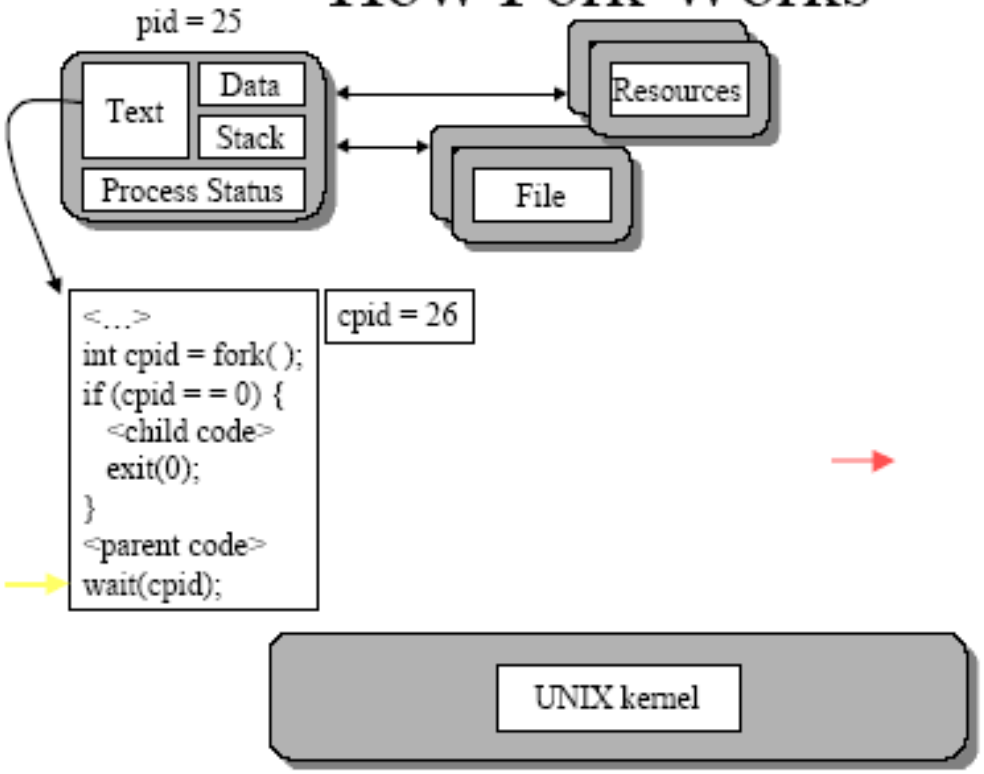
How Fork Works



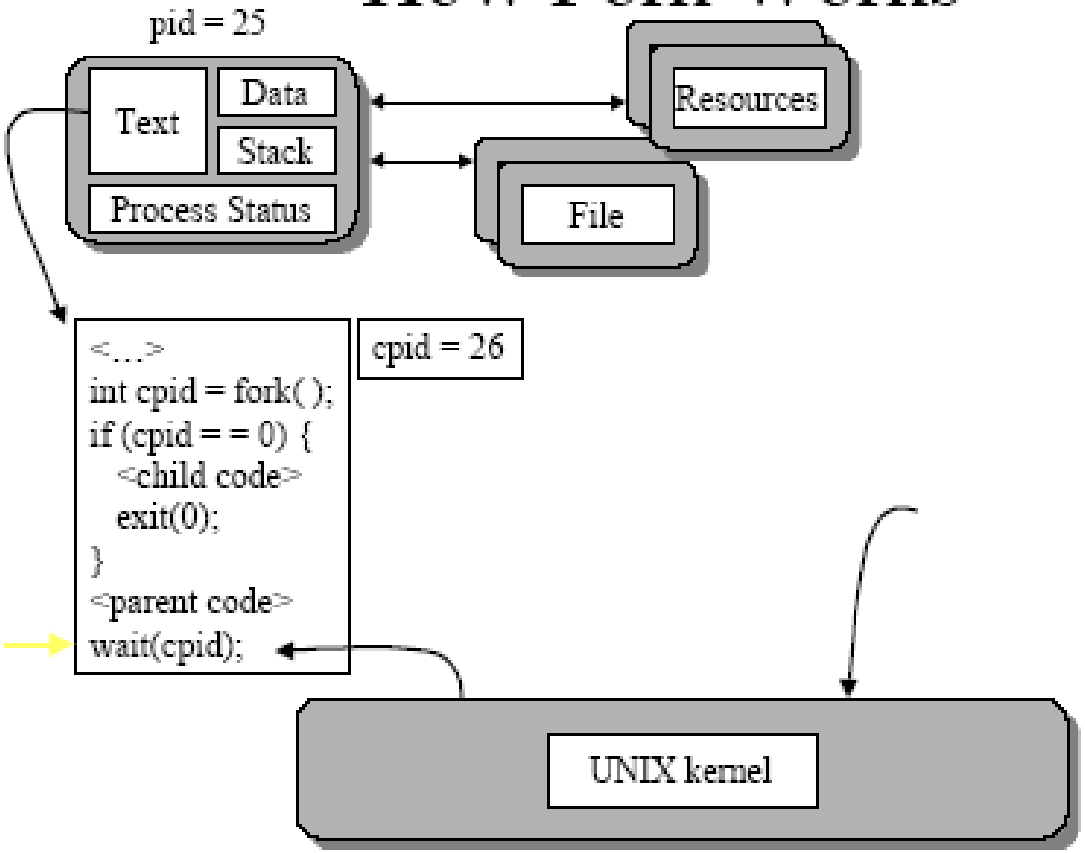
How Fork Works



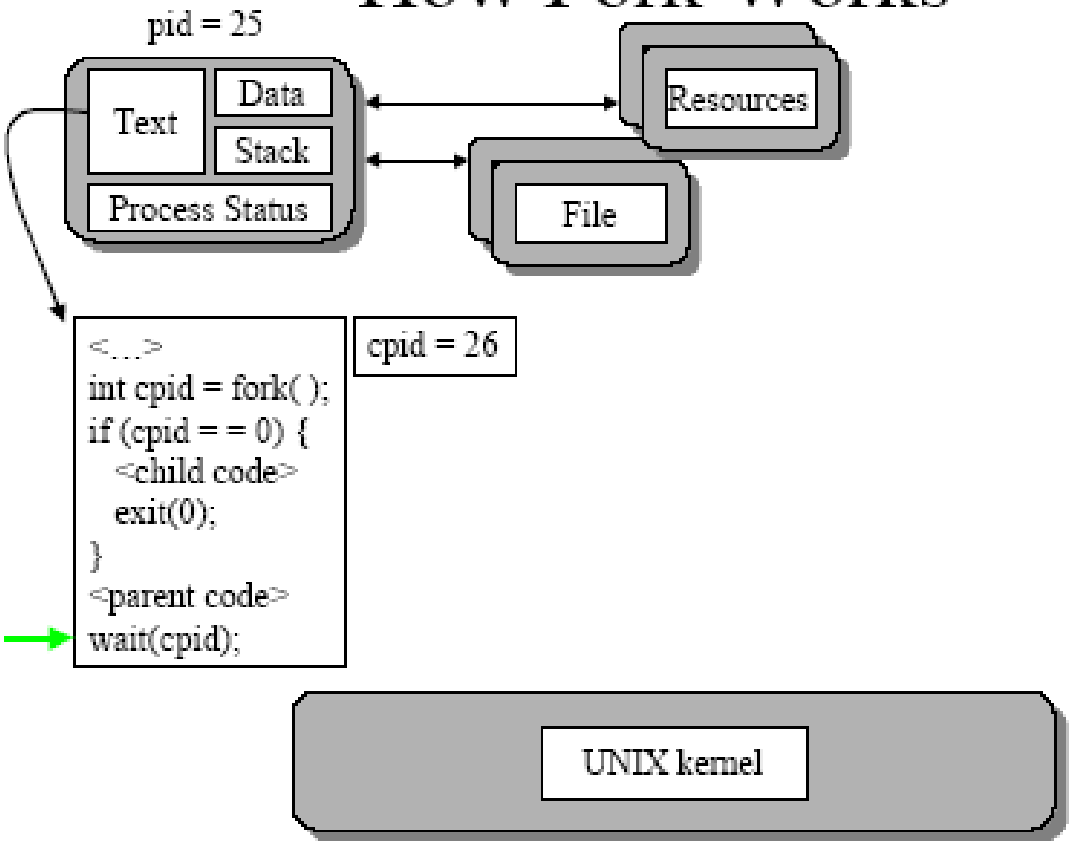
How Fork Works



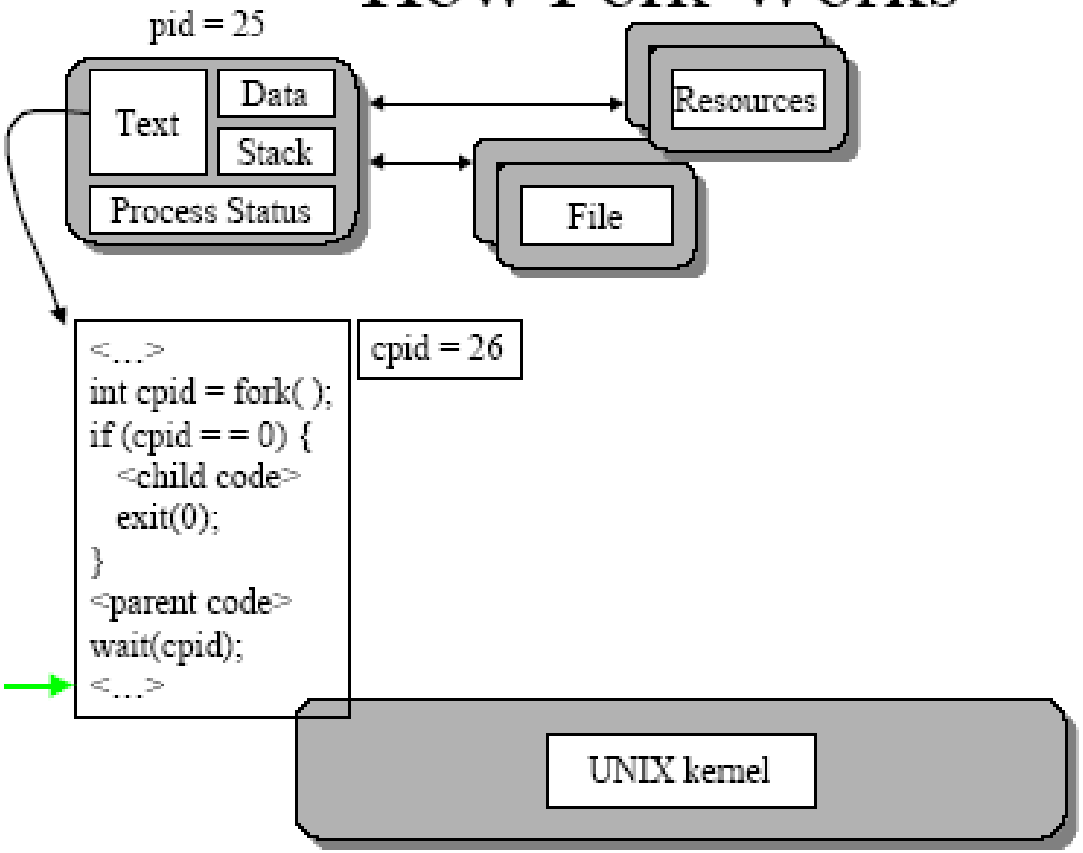
How Fork Works



How Fork Works



How Fork Works



Полнење на програмата – exec()

- **exec**: execl, execle, execlp, execv, execve, execvp, или exect
- **exec** потпрограмата, во сите нејзини форми, извршува нов програм во повикувачкиот процес.
- **exec** потпрограмата не создава нов процес, туку го препокрива тековниот програм со нов (new-process image)
- Тој овозможува процесот да зададе аргументи (argc) и низа стрингови (argv)

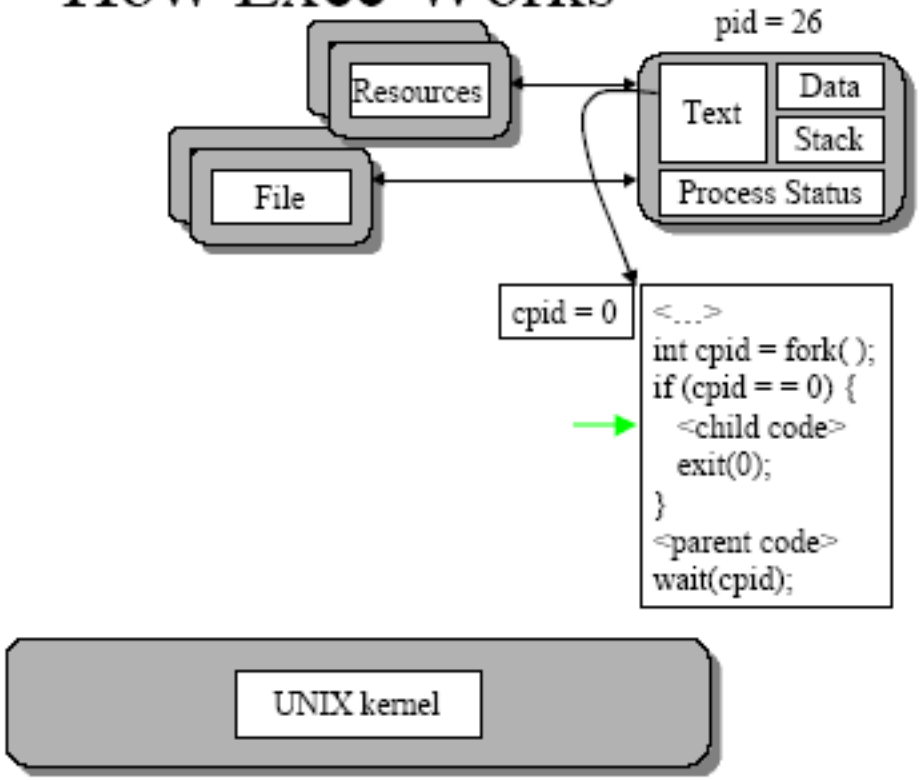
Користење на exec() фамилијата

- Во процесот – татко:

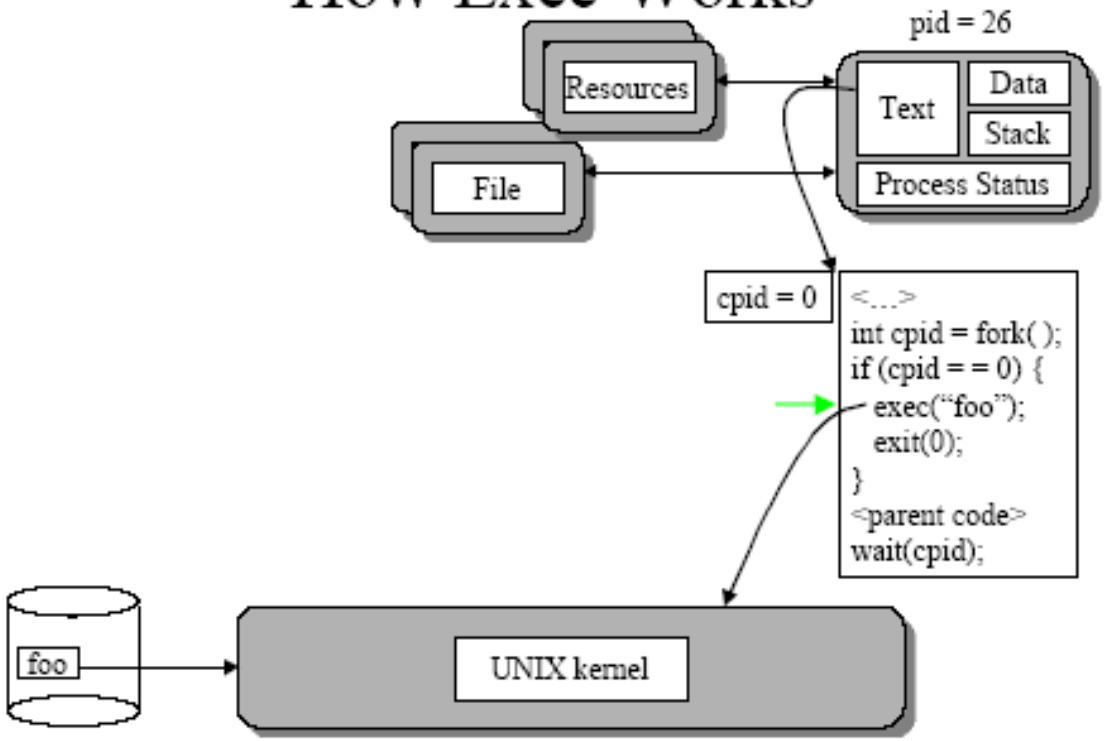
```
main()
…
int pid = fork(); // create a child
if (pid == 0) { // child continues here
exec("program", argc, argv0, argv1, …);
}
else {
   // parent continues here
…
}
```

- Во 99% случаи, ние користиме exec() по повикот fork()

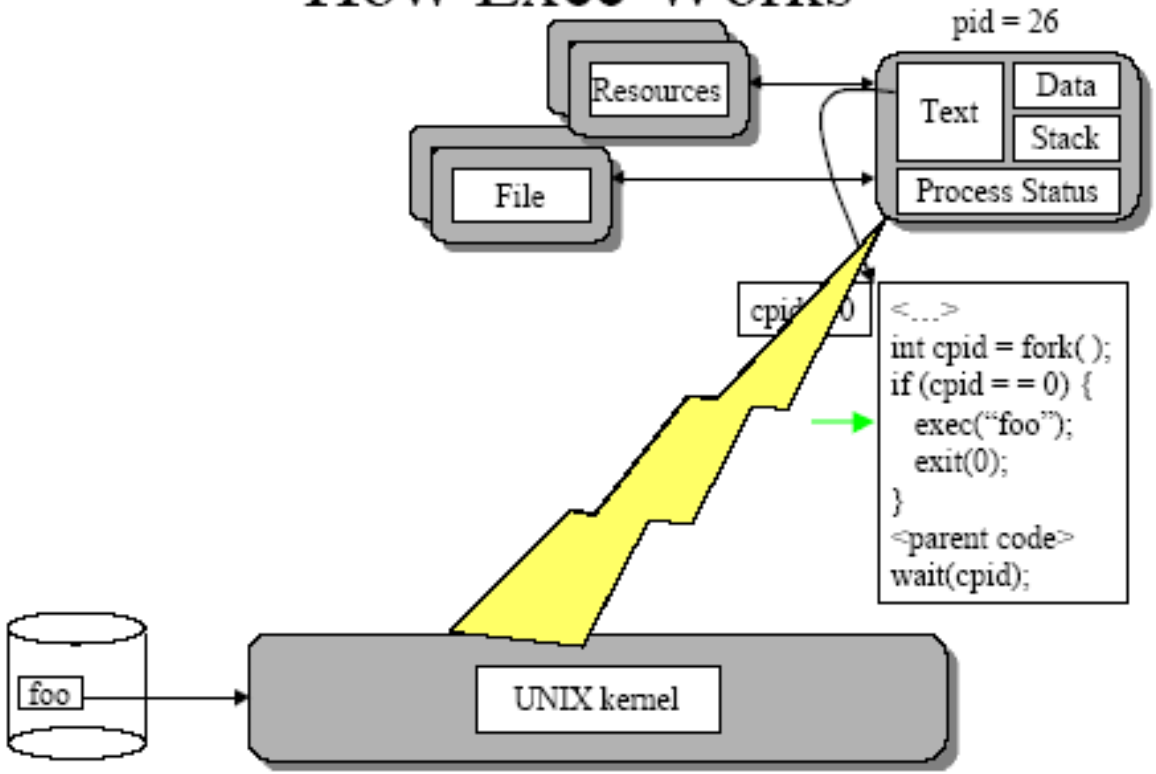
How Exec Works



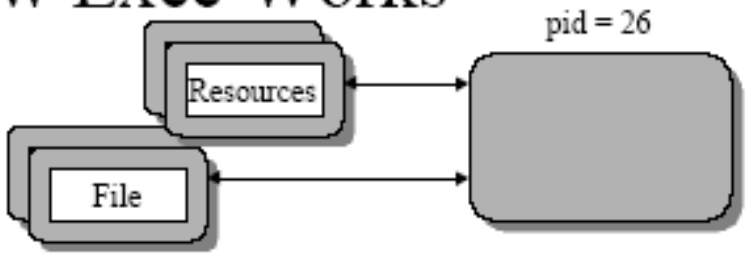
How Exec Works

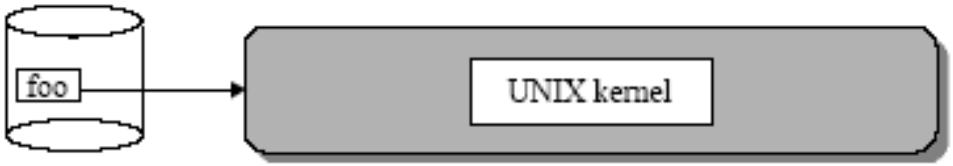


How Exec Works

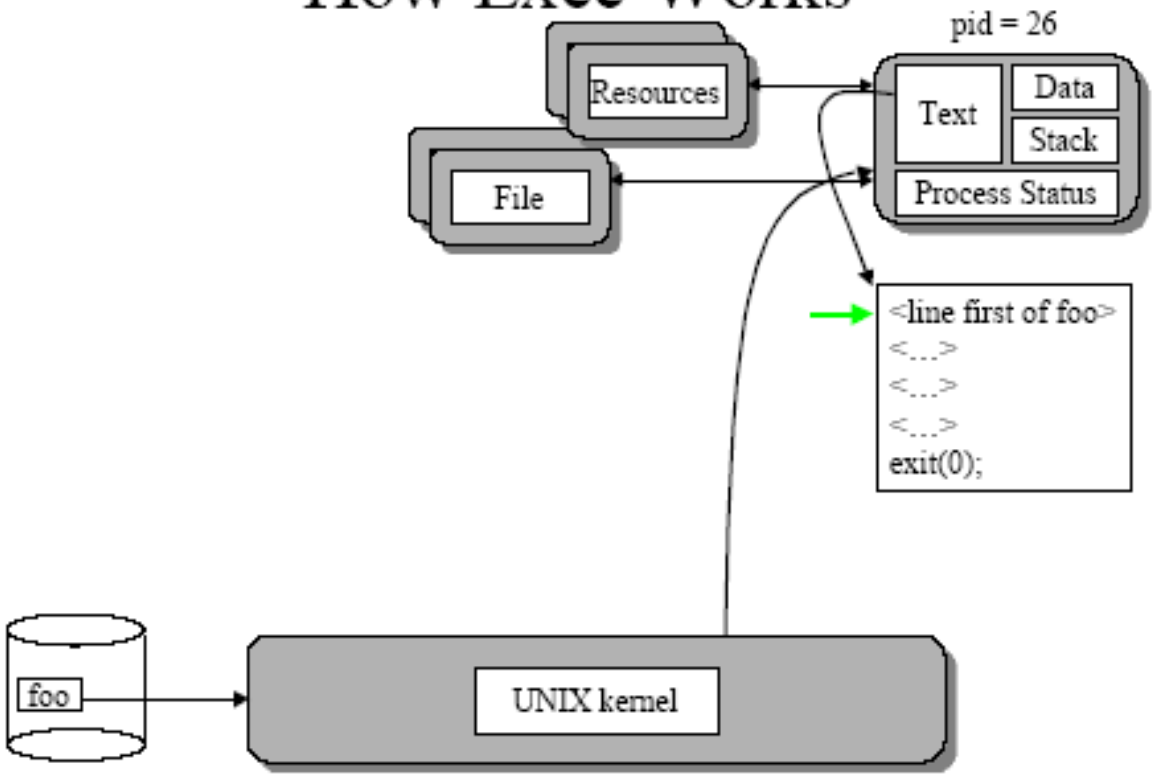


How Exec Works

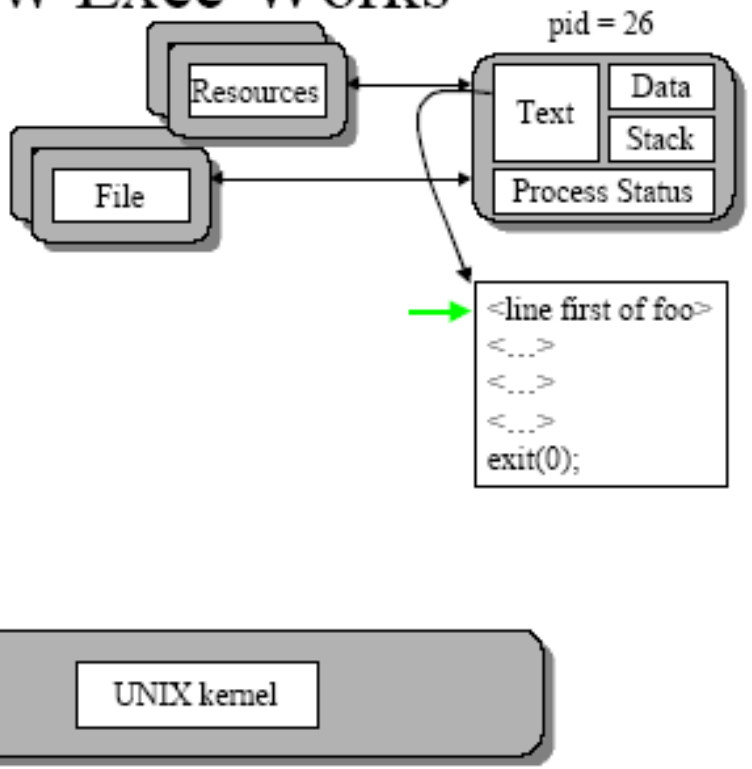




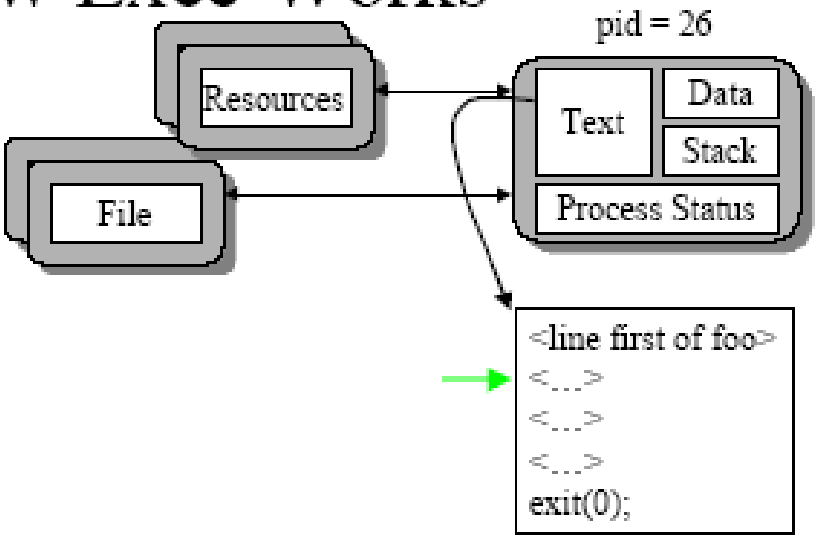
How Exec Works



How Exec Works



How Exec Works



Нормално завршување exit()

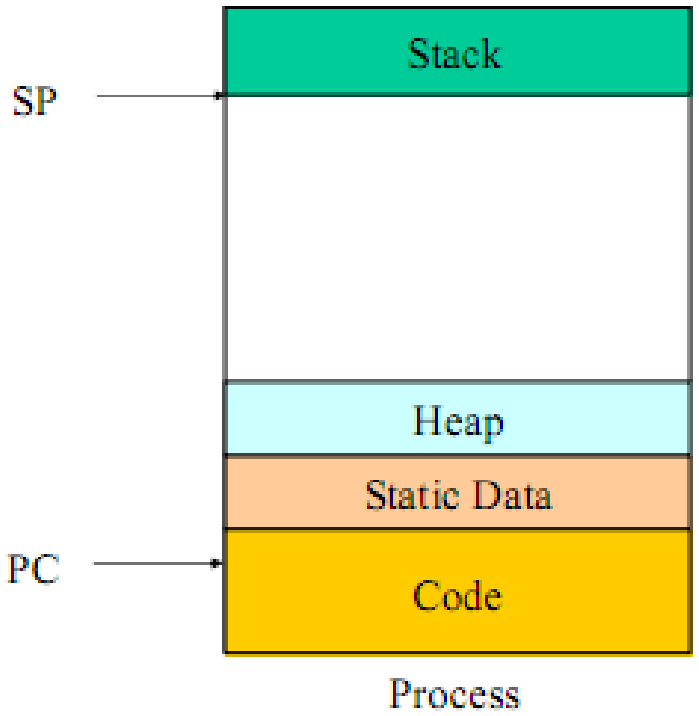
- По завршувањето програмата извршува системски повик *exit*()
- Овој системски повик:
  - го зема “резултатот” вратен од програмата како аргумент,
  - ги затвора сите отворени датотеки, линкови итн.
  - ја деалоцира меморијата
  - ги деалоцира повеќето од структурите на ОС што го поддржувале процесот
  - проверува дали татко- процесот е жив:
- Процесот – дете ја чува резултантната вредност додека таткото не ја побара, не умира туку влегува во zombie/defunct статус

Карактеристики за процеси

- Процесите имаат две карактеристики:
  - **Поседуваат ресурси**
  - **Тек на извршување** – следи тек (thread) на извршување
- Овие две карактеристики се третираат независно од ОС

Процес

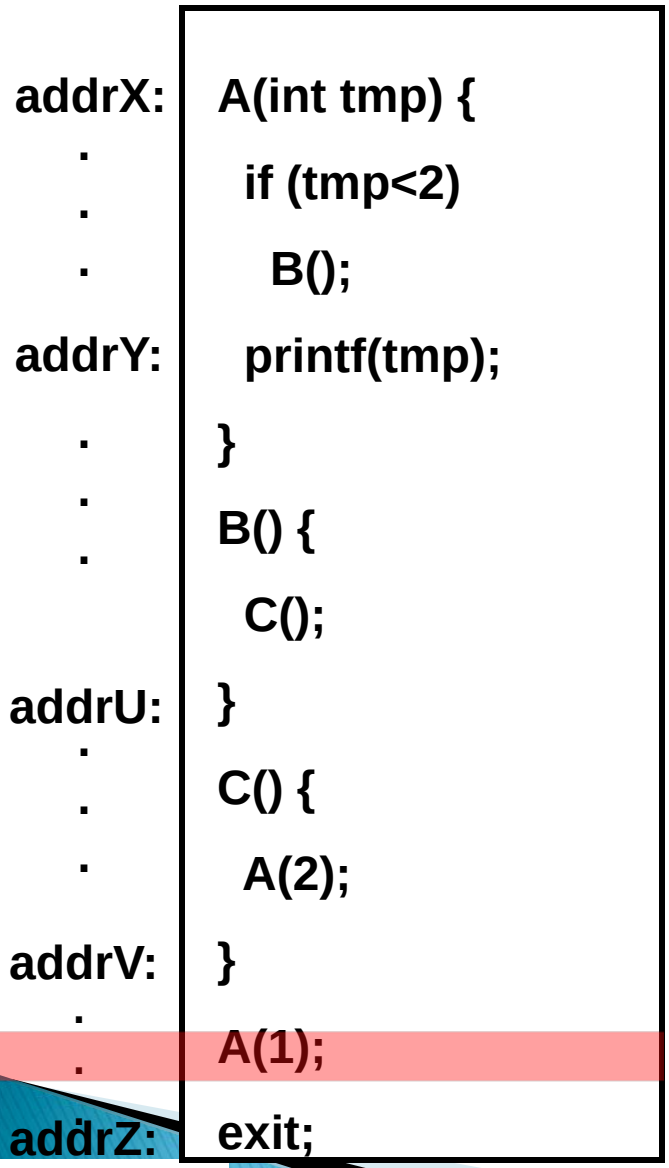
- Извршувачки контекст
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- Код
- Податоци
- Стек



Нишки

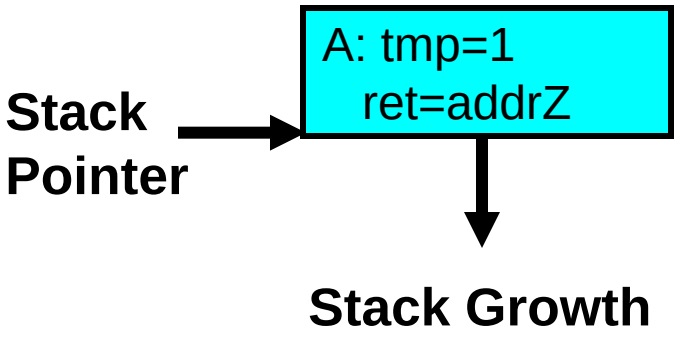
- Паралелизам во рамките на даден процес (Multithreading)
- Паралелно извршување во **ист адресен простор**
- Кооперативност меѓу нишките (не постојат системски механизми за заштита)

Пример за стек на нишка

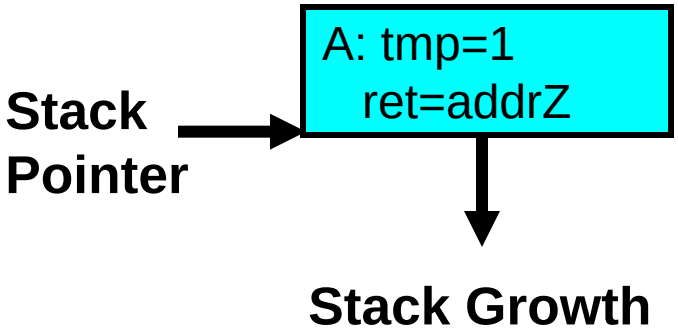


- Се чуваат привремени резултати
- Овозможува рекурзија
- Значајно за модерните јазици

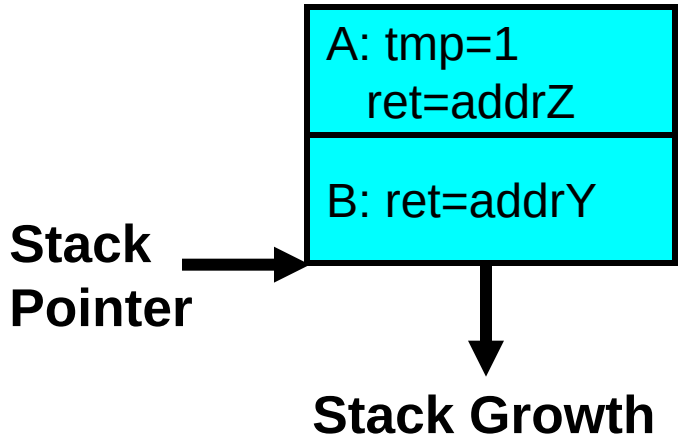
```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



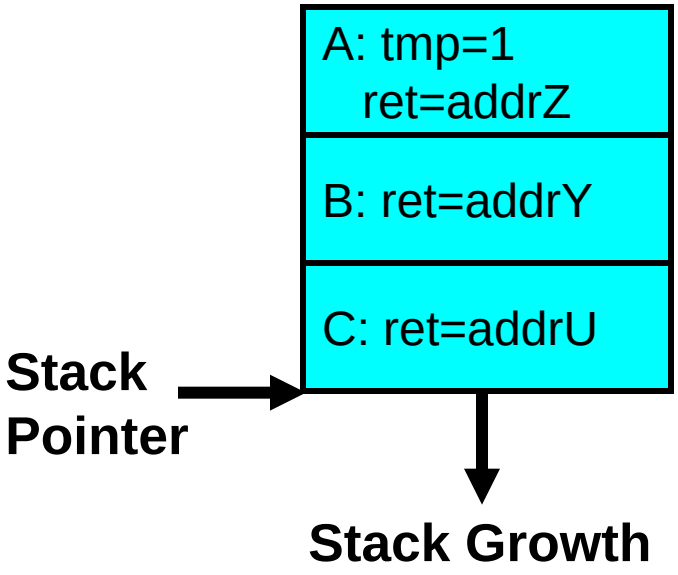
```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



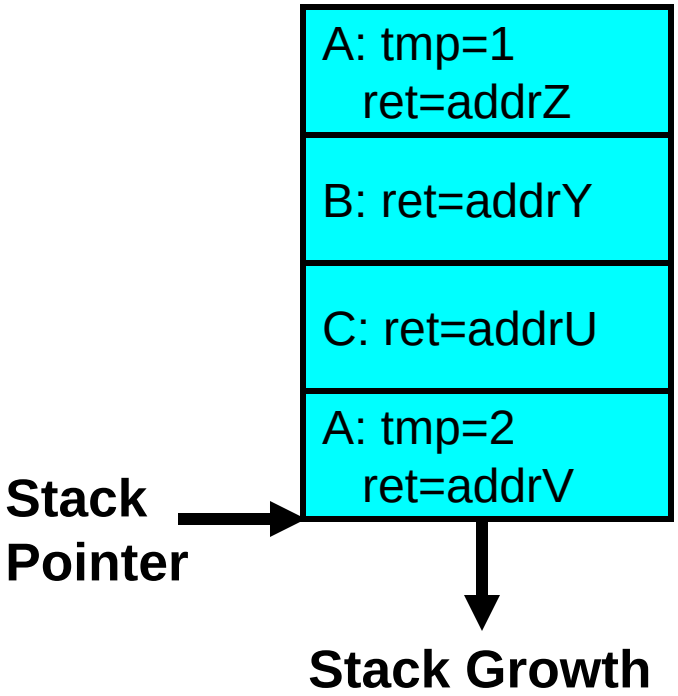
```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



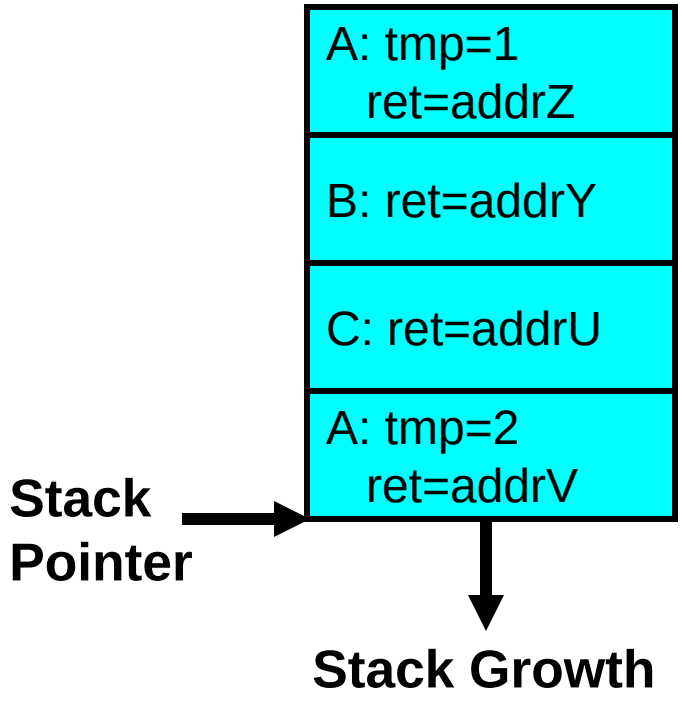
```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



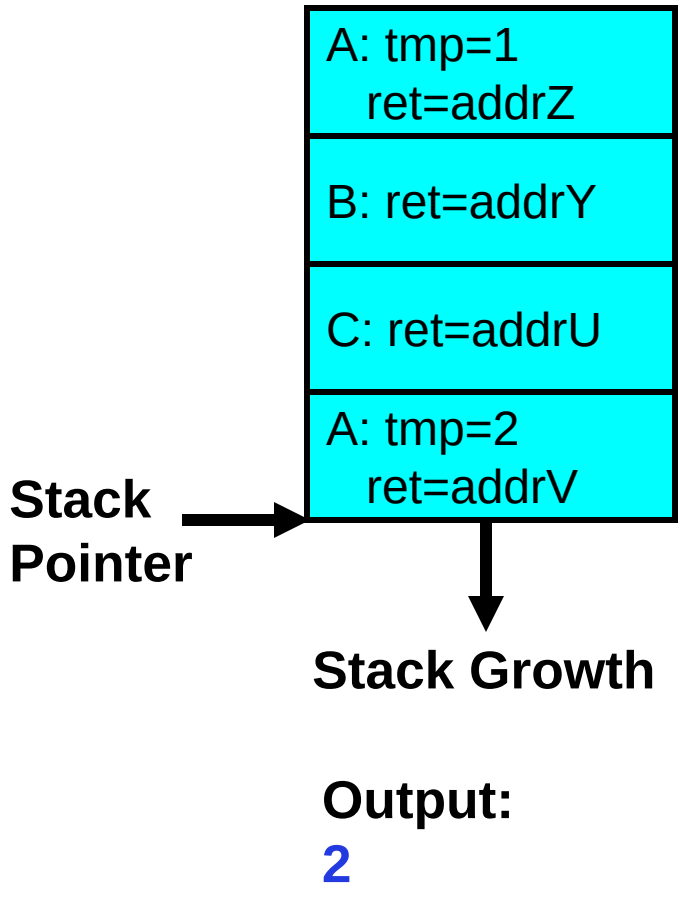
```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



```
addrX:  A(int tmp) {
          if (tmp<2)
            B();
addrY:    printf(tmp);
        }
        B() {
          C();
addrU:  }
        C() {
          A(2);
addrV:  }
        A(1);
addrZ:  exit;
```

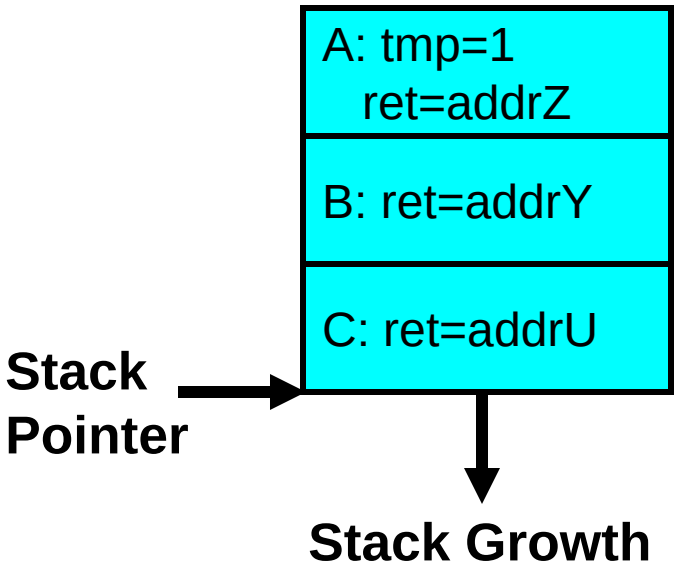


```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



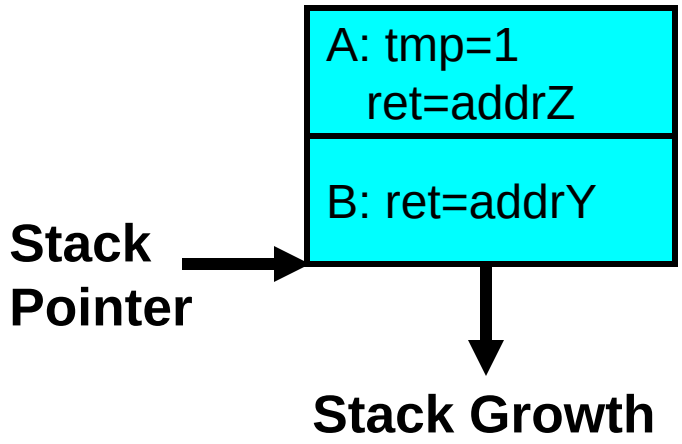
Output:
2

```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



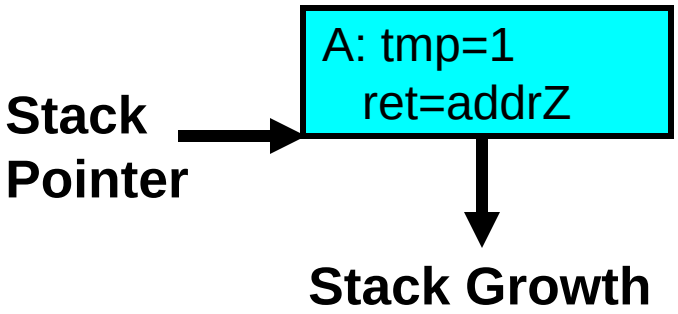
Output:
2

```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



Output:
2

```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```



Output:

2

1

```
addrX:   A(int tmp) {
           if (tmp<2)
             B();
addrY:     printf(tmp);
         }
         B() {
           C();
addrU:   }
         C() {
           A(2);
addrV:   }
         A(1);
addrZ:   exit;
```

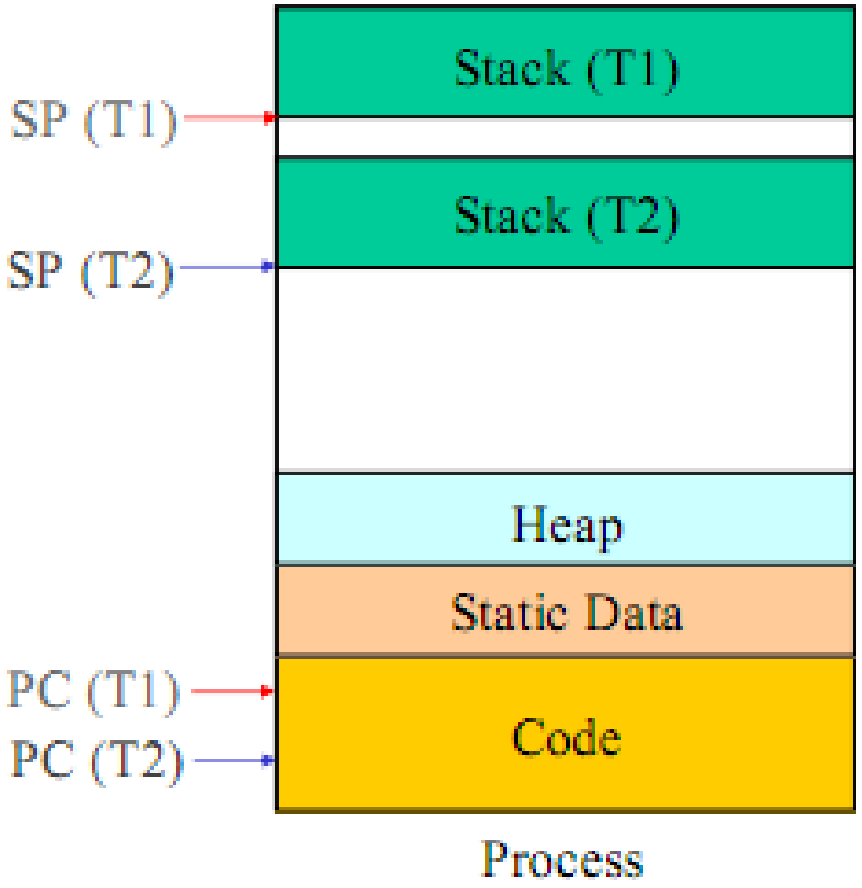
Output:

2

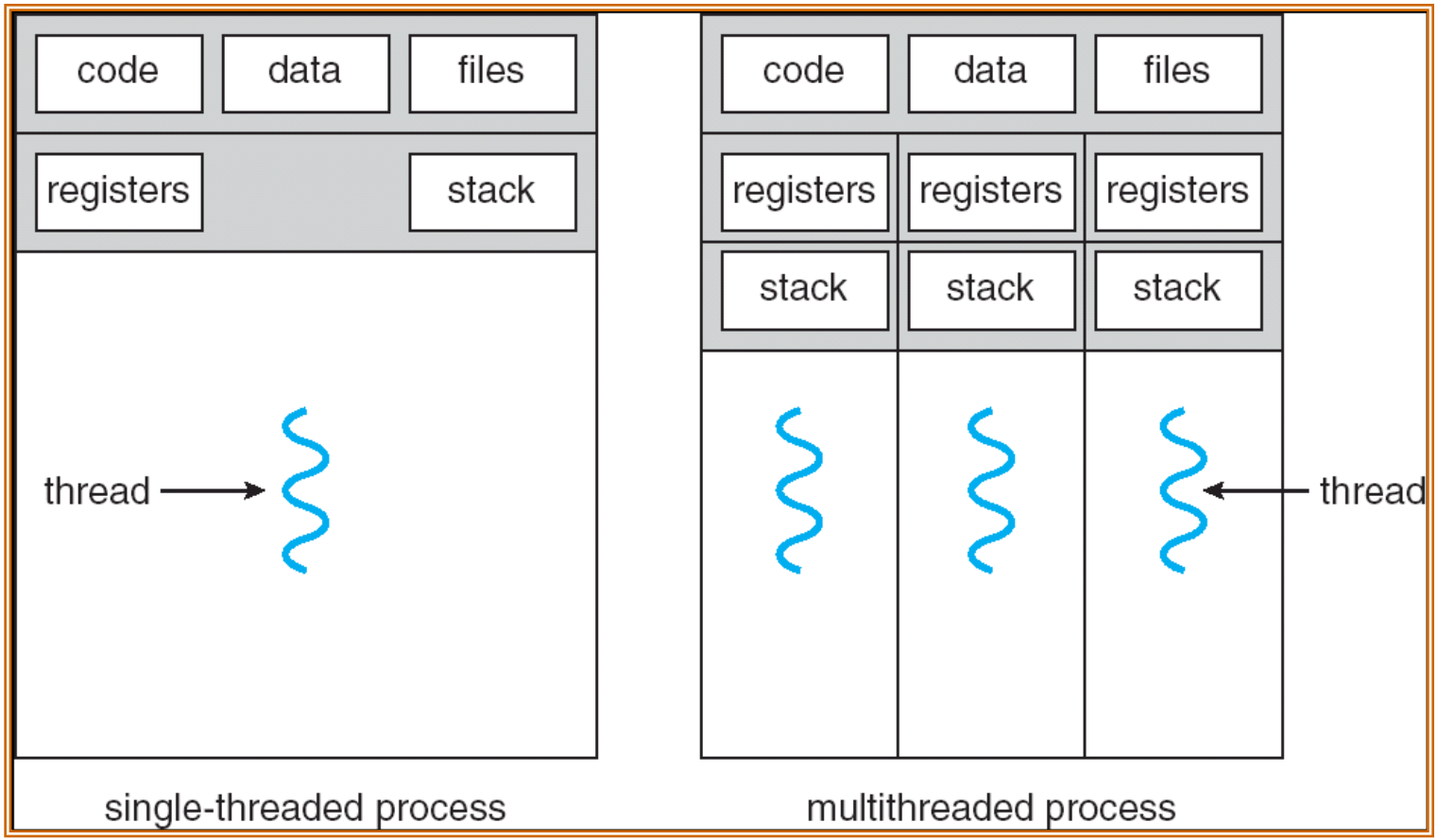
1

Процес со повеќе нишки

- Извршувачки контекст
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers



Процеси со една и со повеќе нишки



Оттука

- Процесите ги групираат ресурсите
- Нишките се нивни извршувачки подентитети кои се распоредуваат за извршување од CPU

Повеќе нишки (Multithreading)

- Е можност на ОС да поддржи повеќе и конкурентни текови на извршување додека сме во истиот процес

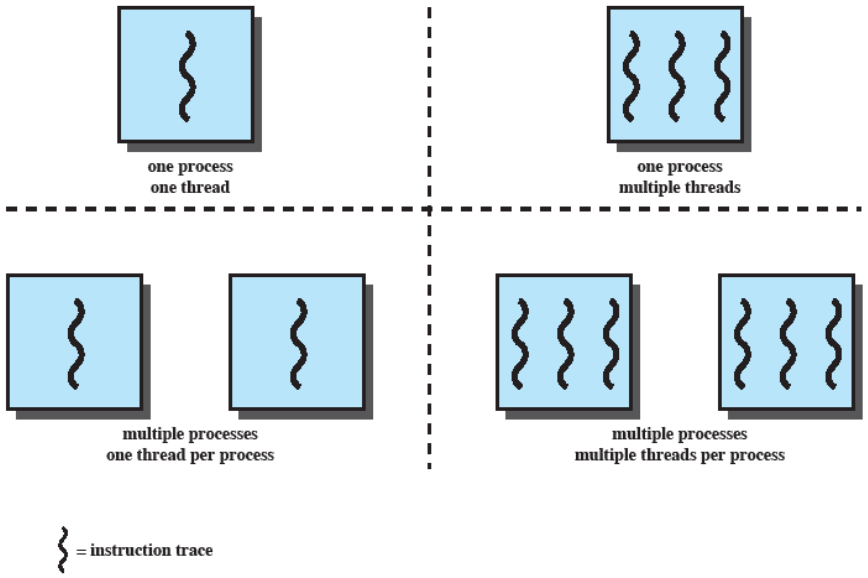


Figure 4.1 Threads and Processes [ANDE97]

Пристапи со една нишка

- MS-DOS поддржуваше по еден кориснички процес до по една нишка
- Некои стари UNIX, поддржуваат повеќе кориснички процеси, но само по една нишка по процес

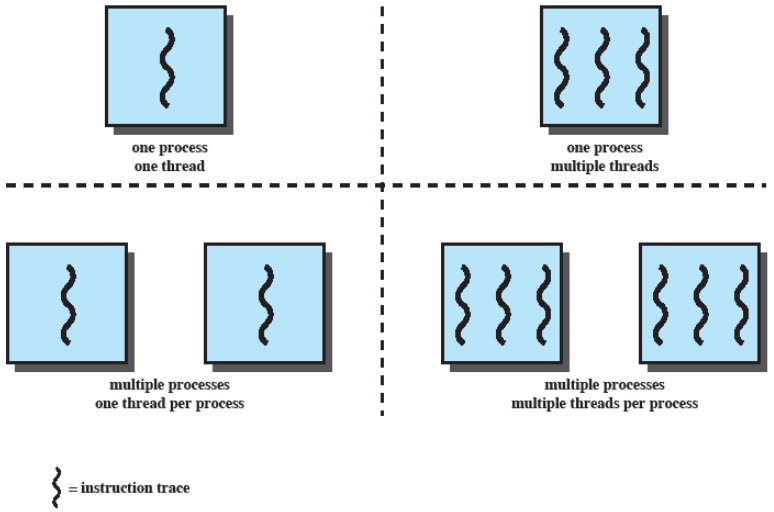


Figure 4.1 Threads and Processes [ANDE97]

Повеќе нишки

- Java run-time околината е едно процесна, но со повеќе нишки
- Повеќекратни процеси и нитки се наоѓаат во Windows, Solaris, и многу модерни верзии на UNIX

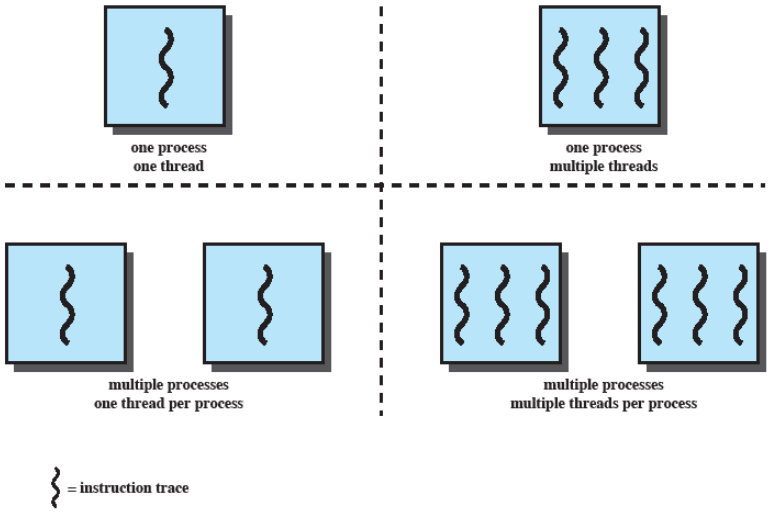


Figure 4.1 Threads and Processes [ANDE97]

Предности на нишките

- Поедноставно (побрзо) се создаваат и уништуваат отколку процесите
- Менувањето нишки е побрзо од менување процеси
- Нишките комуницираат меѓу себе без потреба од јадро
- Забрзување на апликациите – во ситуации кога има многу процесорска активност, а истовремено и значителни влезно-излезни процеси
- Можна вистинска паралелност – кај системи со повеќе CPU

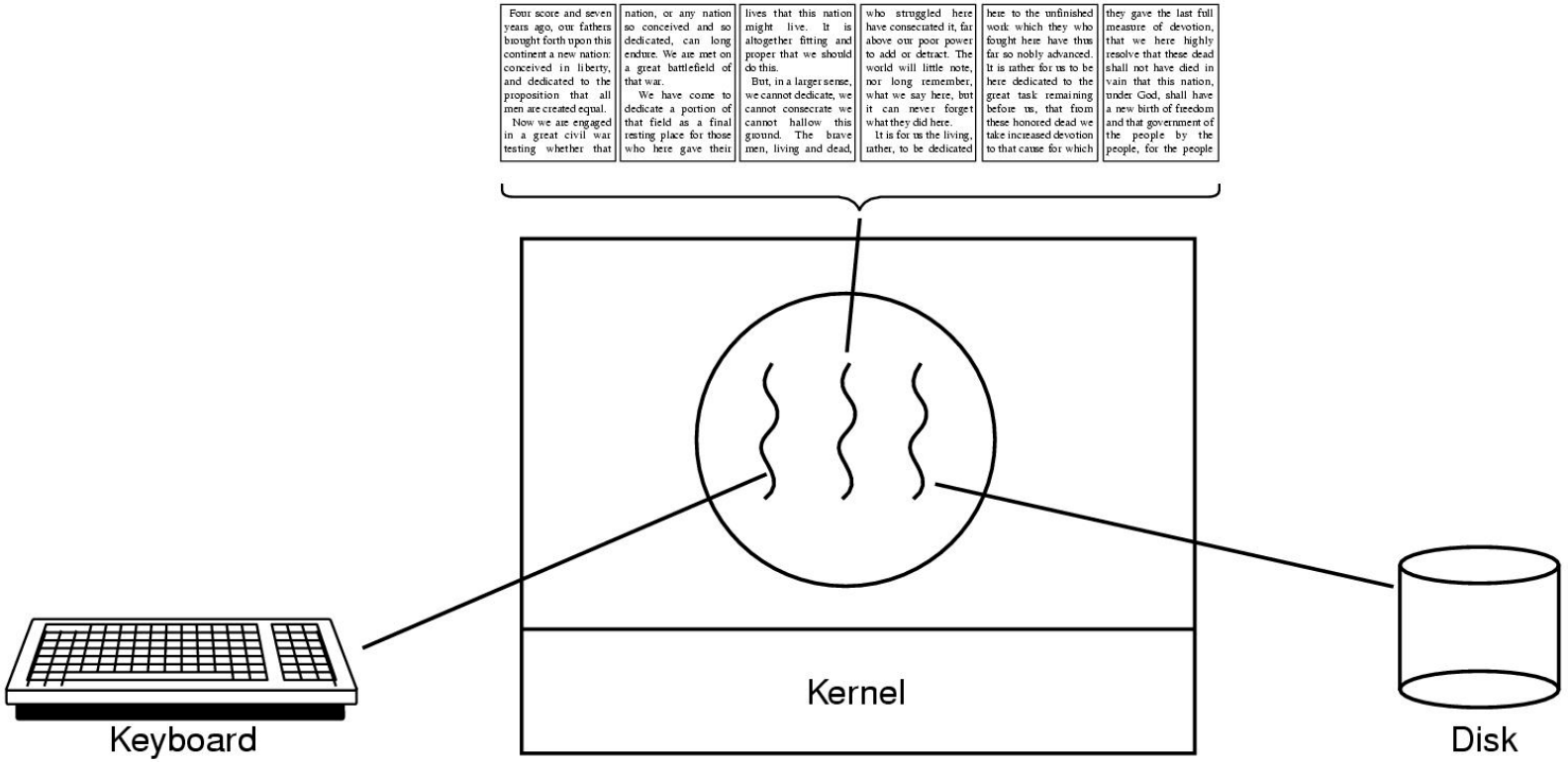
Пример 1 – пишување книга

- Нека корисникот пишува книга и да претпоставиме дека во даден момент е потребно да се изврши промена на текстот
- Доколку има само еден процес (со една нитка), тогаш потребно е во исто време да се следат
  - Командите од корисникот (од тастатура и глушец)
  - Преформатирање на текстот
  - Снимање на новата содржина на диск
- Решение со процеси: Сложен програмски модел базиран на прекини!!!

Пример 1 – пишување книга

- Подобро решение
- Да имаме процес со три нишки
  1. за интеракција со корисникот
  2. за преформатирање на текстот по потреба
  3. за запишување на содржина на дискот
- Трите нишки ја споделуваат заедничката меморија, па така имаат пристап до документот што се обработува.
- Опцијата со три посебни процеси (наместо три нитки) не функционира во овој случај!

Пример 2 – Користење на нишки



Пример 2: апликации кои обработуваат многу податоци

- Вообичаен пристап:
  - Прочитај блок податоци
  - Обработи го
  - Запиши го
- Проблем: Ако се достапни само блокирачки системски повици, процесот се блокира додека влегуваат и излегуваат податоци
- Залудно трошење на ресурсите – CPU без работа!

Пример 2: апликации кои обработуваат многу податоци

- Решение: Нишки
- Процесот може да се структурира со три нишки:
  1. за влез
  2. за обработка
  3. за излез
- Овој модел функционира само ако системскиот повик ја блокира нитката што повикува, а не целиот систем!