

Оперативни Системи

Процеси

проф. д-р Димитар Трајанов,
проф. д-р Невена Ацковска,
проф. д-р Боро Јакимовски,
проф. д-р Весна Димитрова,
проф. проф. д-р Игор Мишковски,
вон. проф. д-р Сашо Граматиков,
вонр. д-р Милош Јовановиќ,
доц. д-р Ристе Стојанов,
Д-р Костадин Мишев



Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО



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

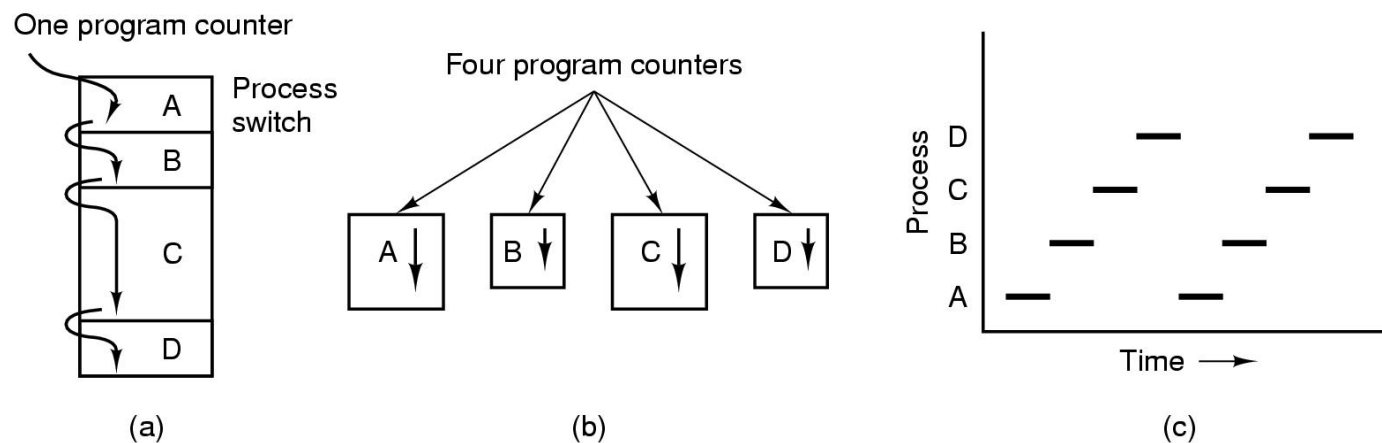
- Во секој миг паралелно работат повеќе процеси
 - ОС стартувани процеси
 - Кориснички стартувани процеси
- Корисникот има илузија дека се извршуваат паралелно



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

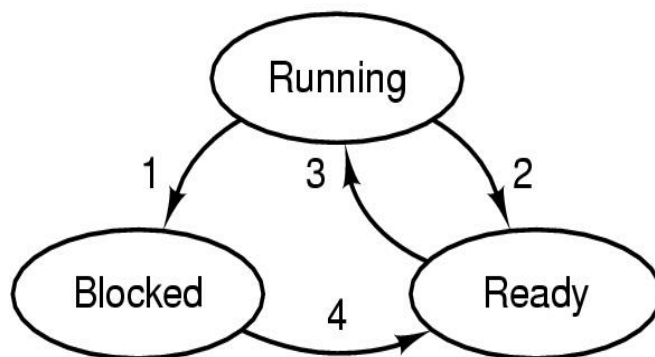
1. Нормален излез (своеволно)
2. Излез поради грешка (своеволно)
3. Фатална грешка (насилно)
4. “Убивање” од друг процес (насилно)



Хиерархија на процеси

- Родителите креираат деца процеси, децата може да креираат свои 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



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

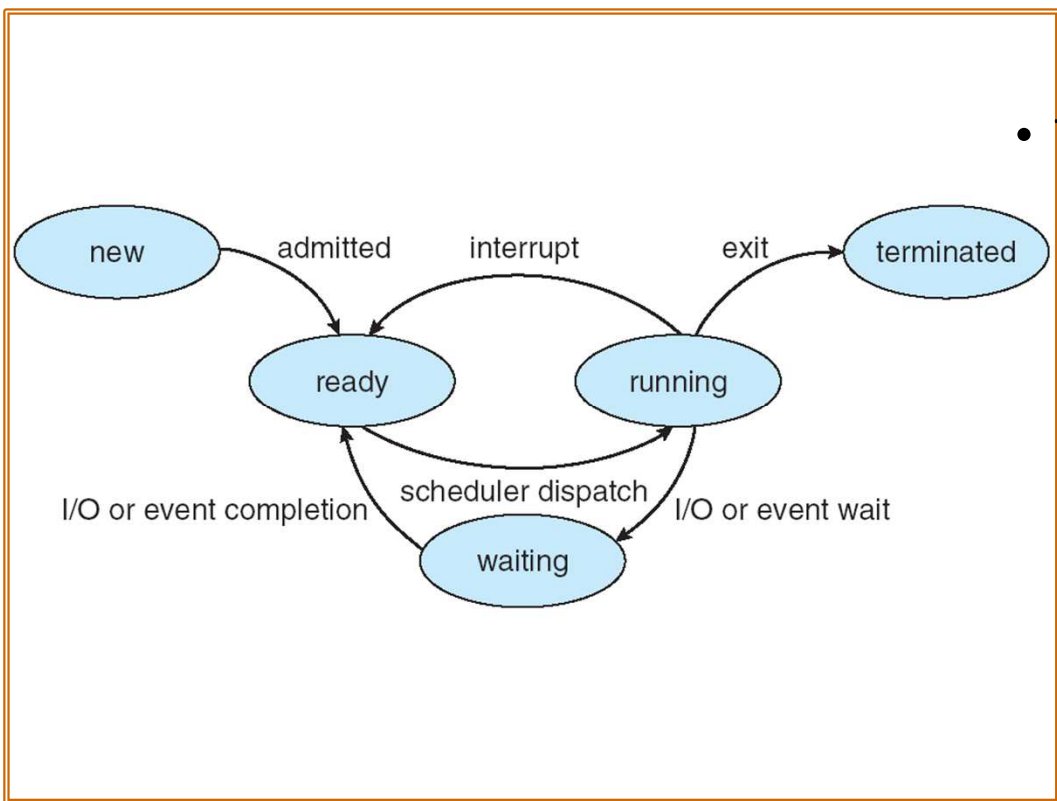
- Можни состојби
 - running
 - blocked
 - ready



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;

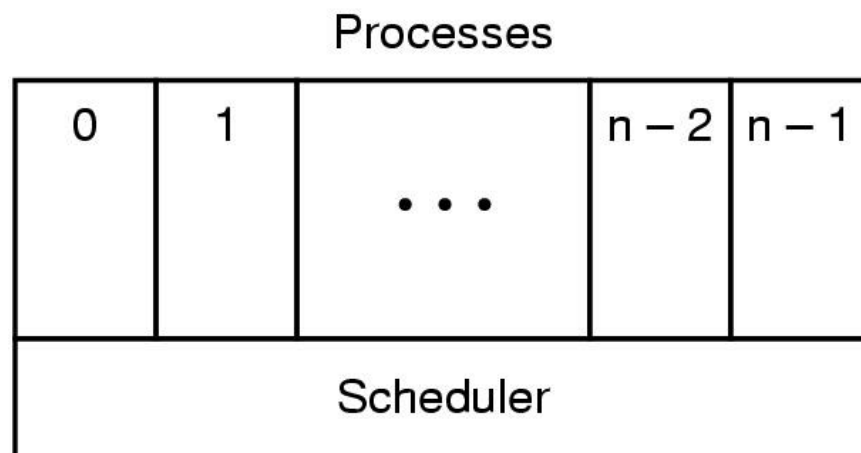
Животен циклус на процес



- 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 РСВ

- **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**
 - показувачи за редовите на чекање
 - приоритет на процес;



Информации во РСВ (2)

- **Информации за управување со меморија**
 - мемориски описи (во зависност од тоа каква меморија користи ОС);
 - Base и limit регистар и табелата на страници
- **Кориснички идентификациони броеви**
 - uid, gid
- **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 Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Полиња во табелата за процеси

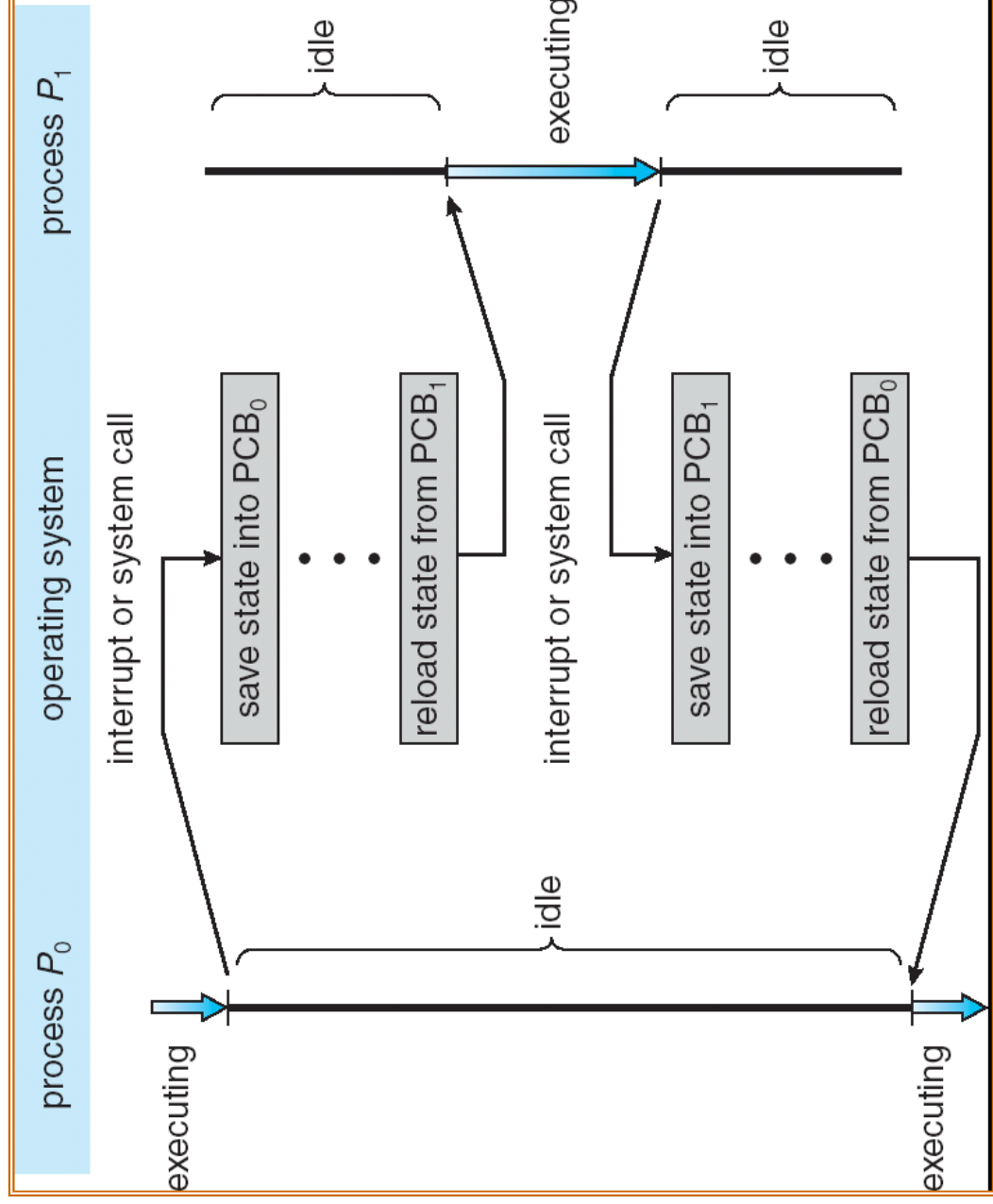
Контекст на процес

- **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;
- 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
- Системски повик
 - отворање на датотека
 - Крај на процес





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 родителот:
 - идентични копии на променливите и меморијата (адресен простор)
 - идентични копии на сите CPU регистри (освен еден)



UNIX-овиот `fork()` (2)

- Двата процеса (родителот и детето) се извршуваат од истата точка по враќањето од повикот `fork()`:
 - за детето - процес `fork()` враќа 0 (PID за детето)
 - за родителот - 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 родителите во детето – процес
 - **скапо!**



Користење на fork()

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

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

```
if (fork() == 0) {
...
}
...
```

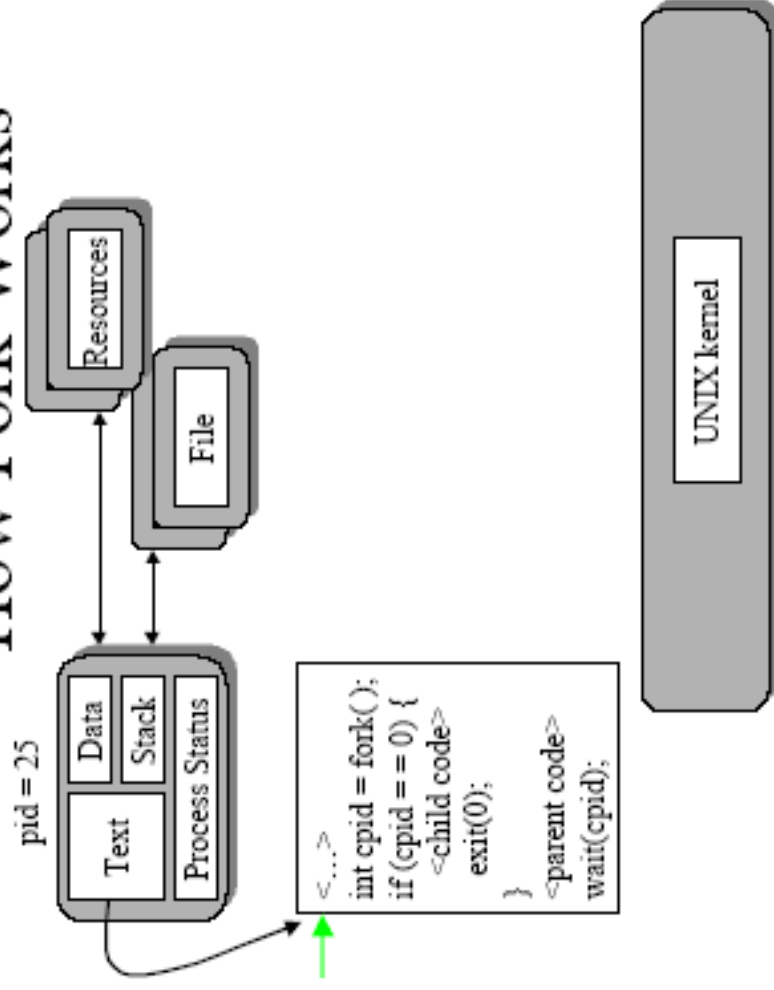
Кодот на родителот и на детето се во иста „програма“

Детето ги наследува сите отворени датотеки и мрежни конекции



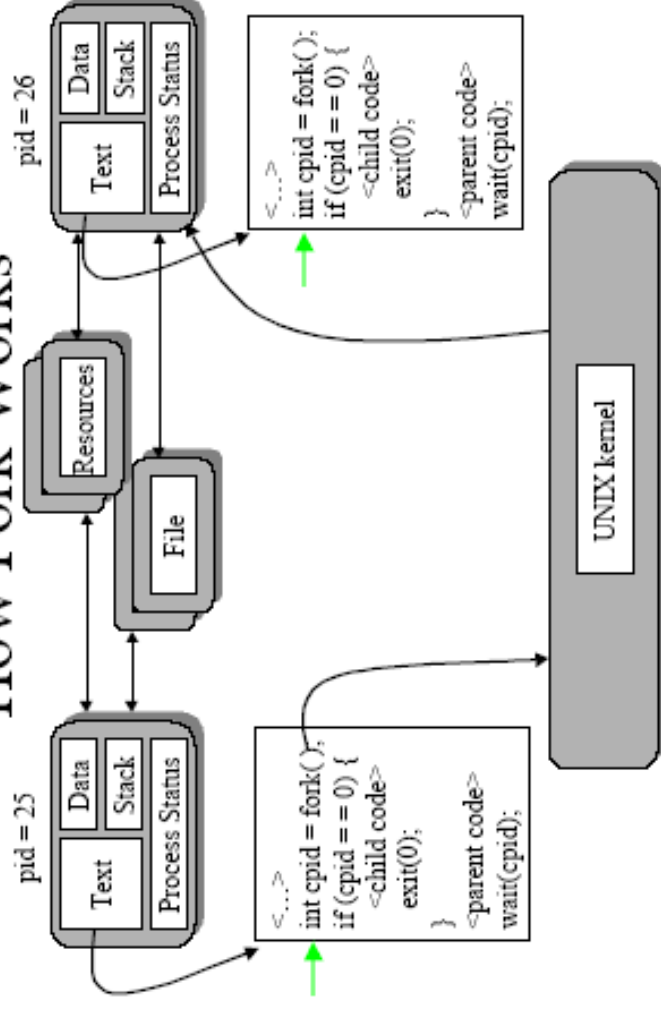


How Fork Works



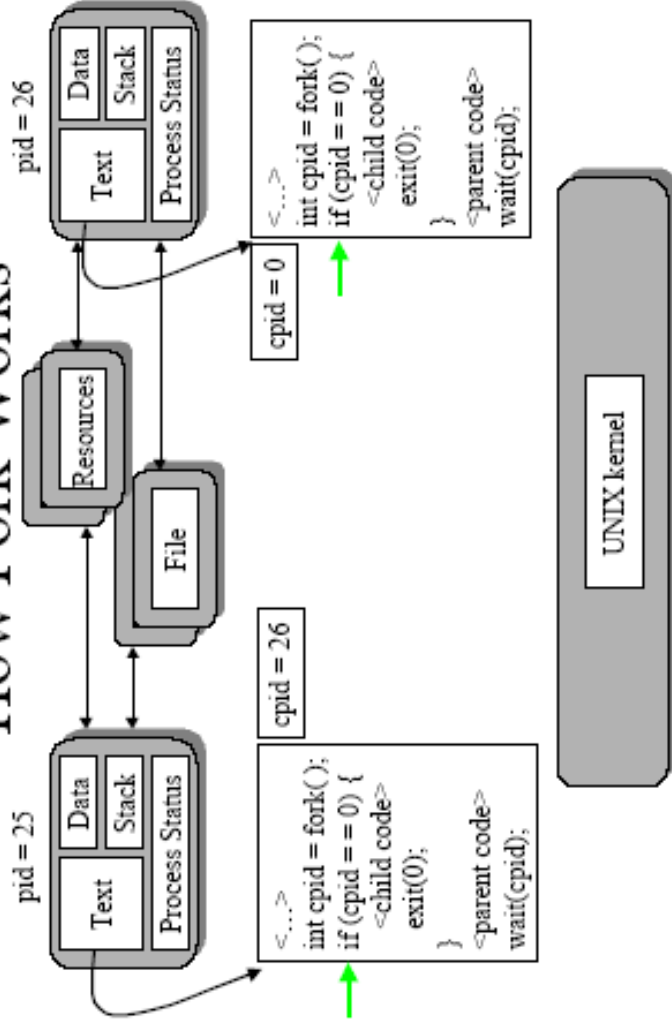


How Fork Works



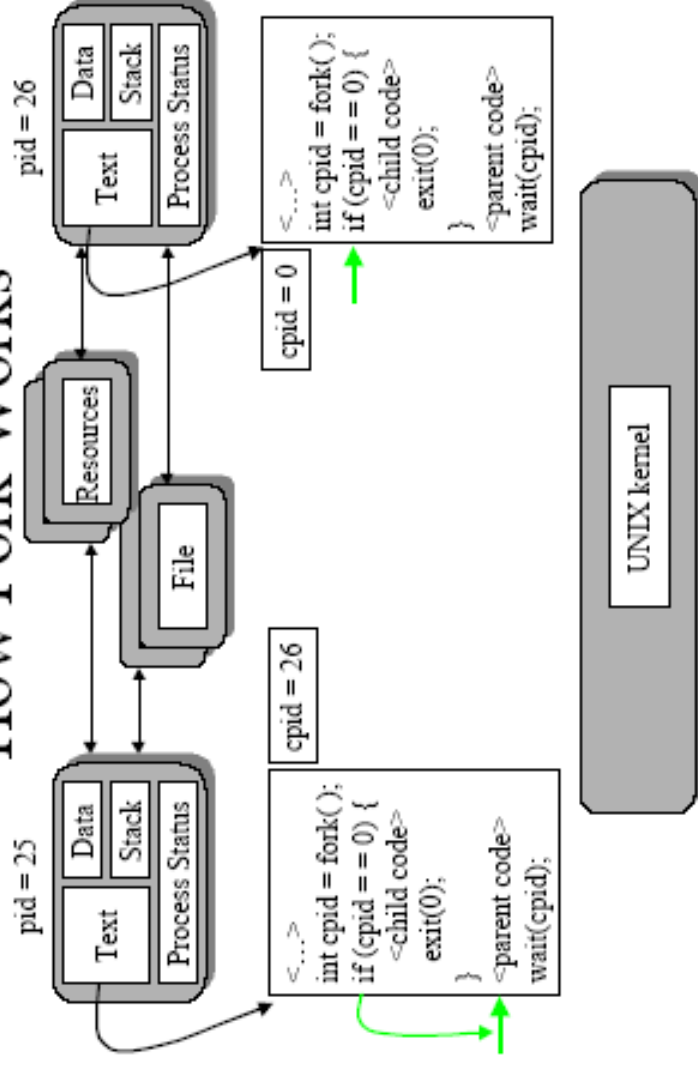


How Fork Works

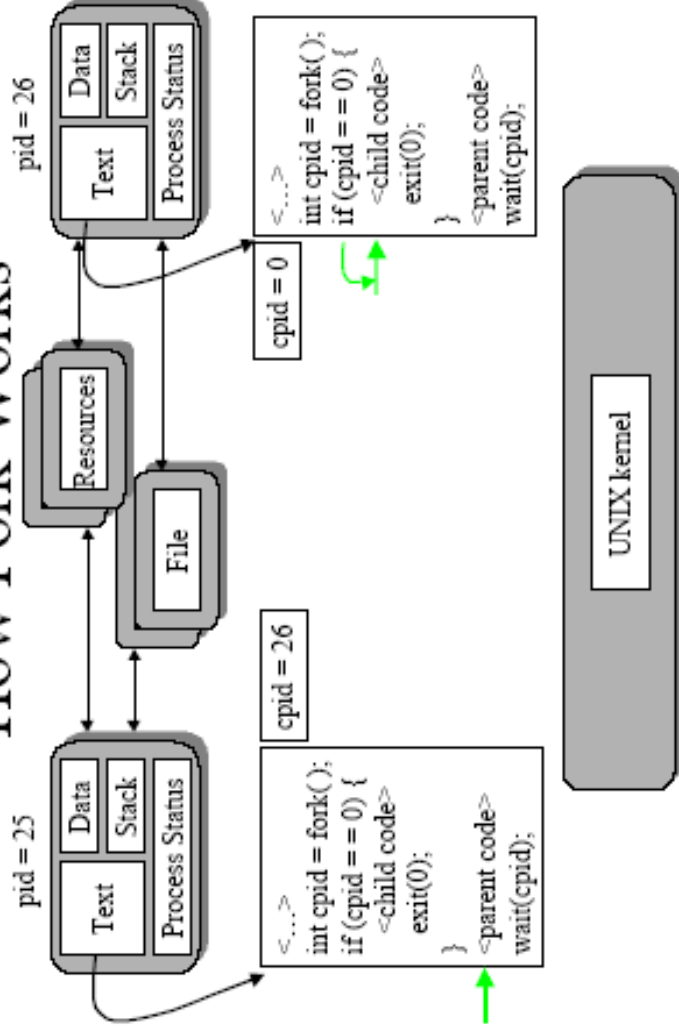




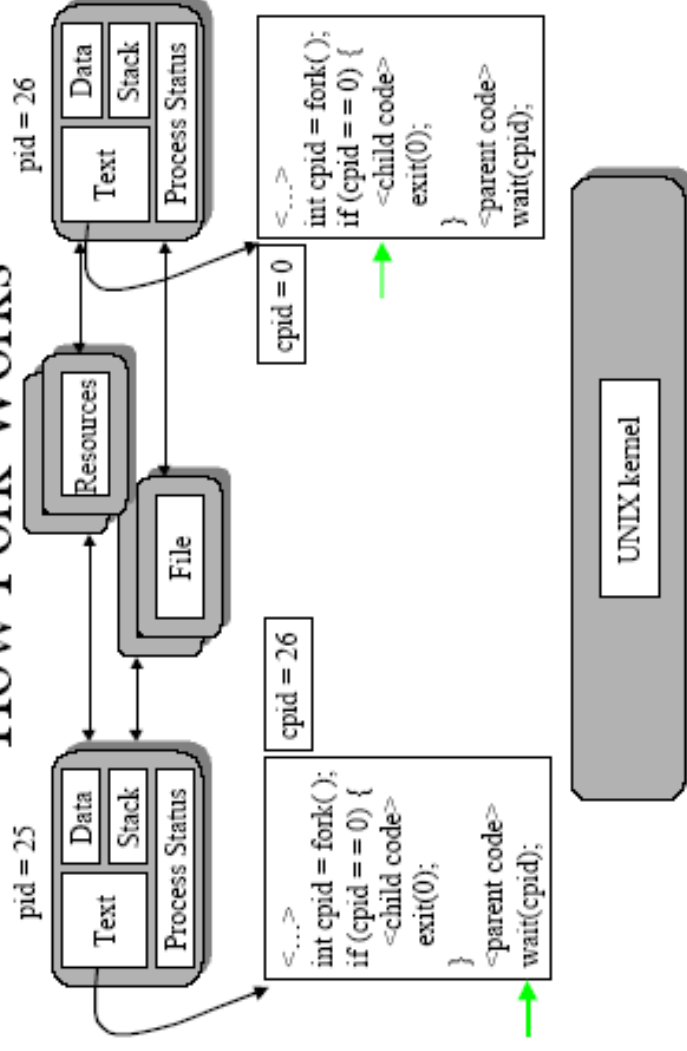
How Fork Works



How Fork Works

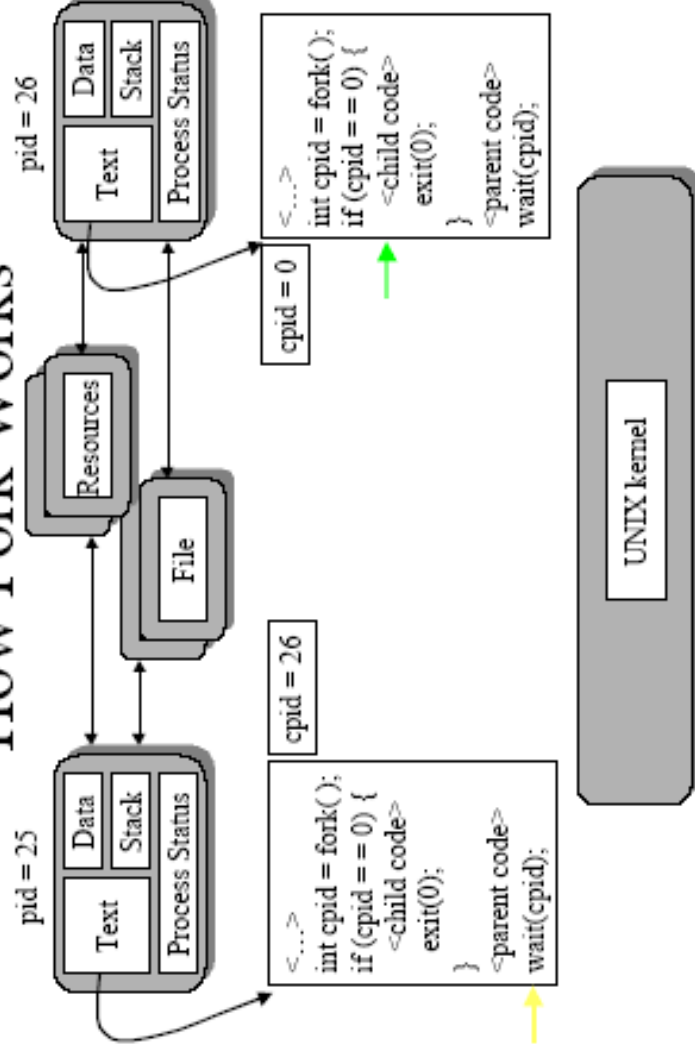


How Fork Works



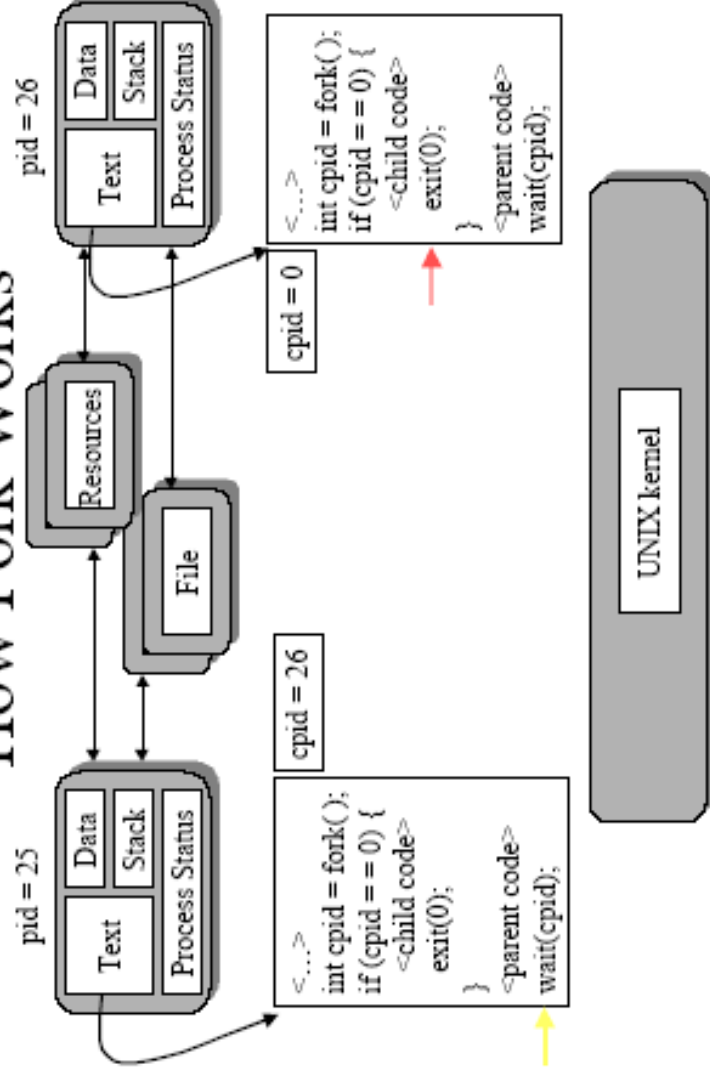


How Fork Works



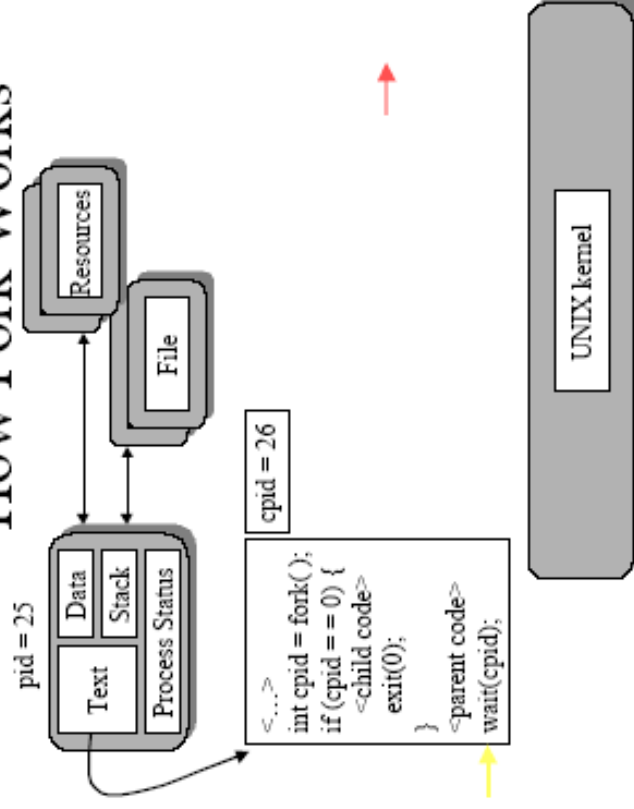


How Fork Works

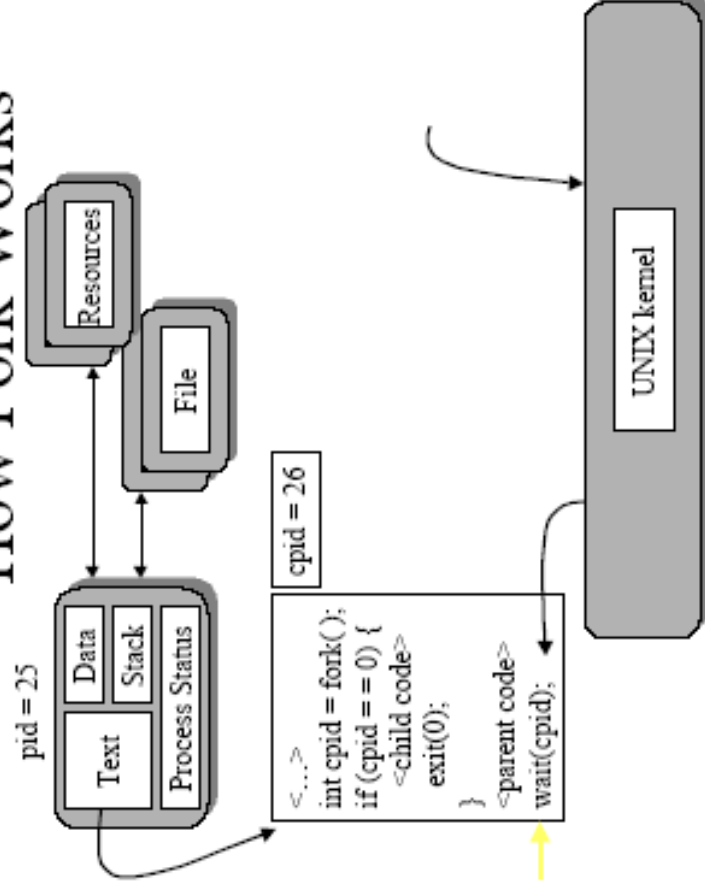




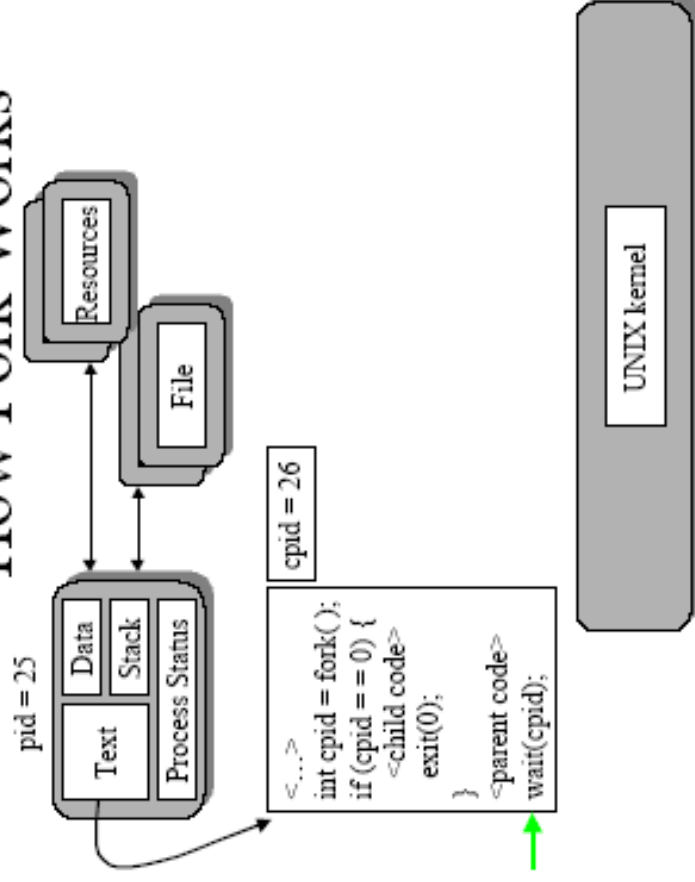
How Fork Works



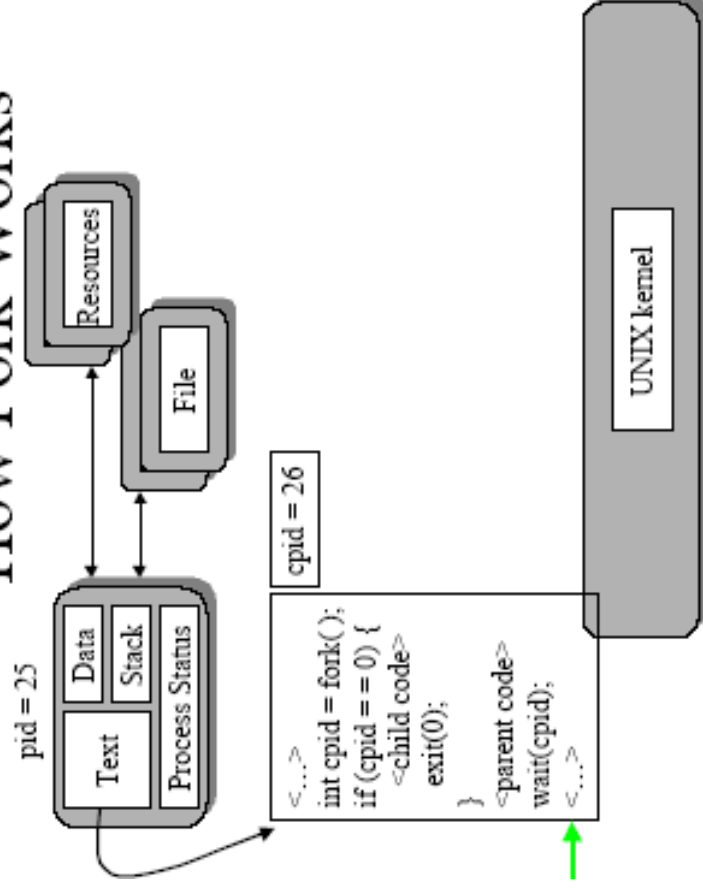
How Fork Works



How Fork Works



How Fork Works



Полнење на програмата - `exec()`

- **exec**: `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`, или `exec`
- **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 - родител:

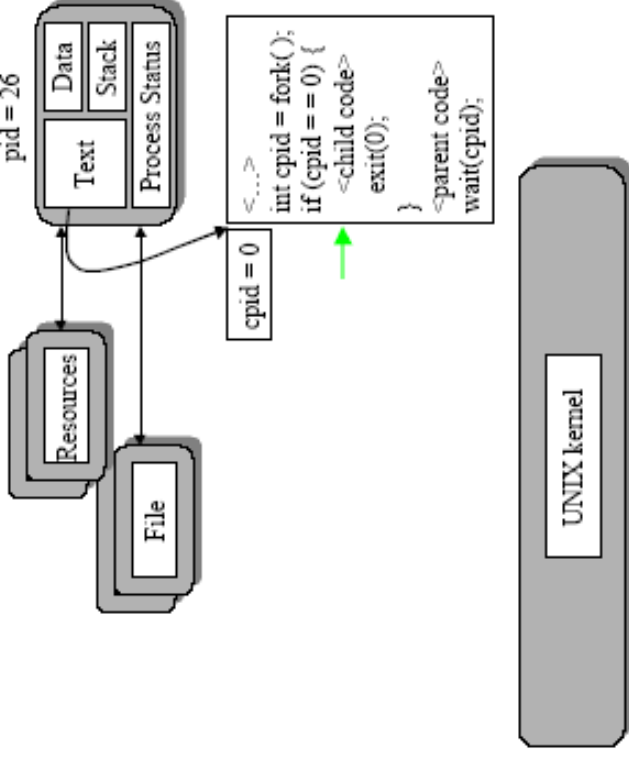
```
main()
...
int pid = fork(); // create a child
if (pid == 0) { // child continues here
    exec("program", argc, argv0, argv1, ...);
}
else {
    // parent continues here
    ...
}
```

- Во 99% случаи, се користи `exec()` по повикот `fork()`



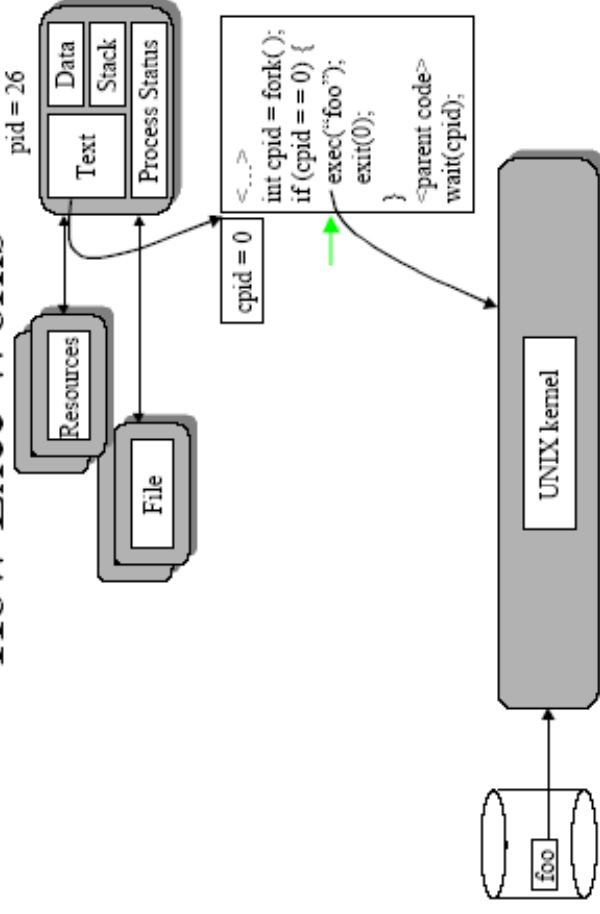


How Exec Works



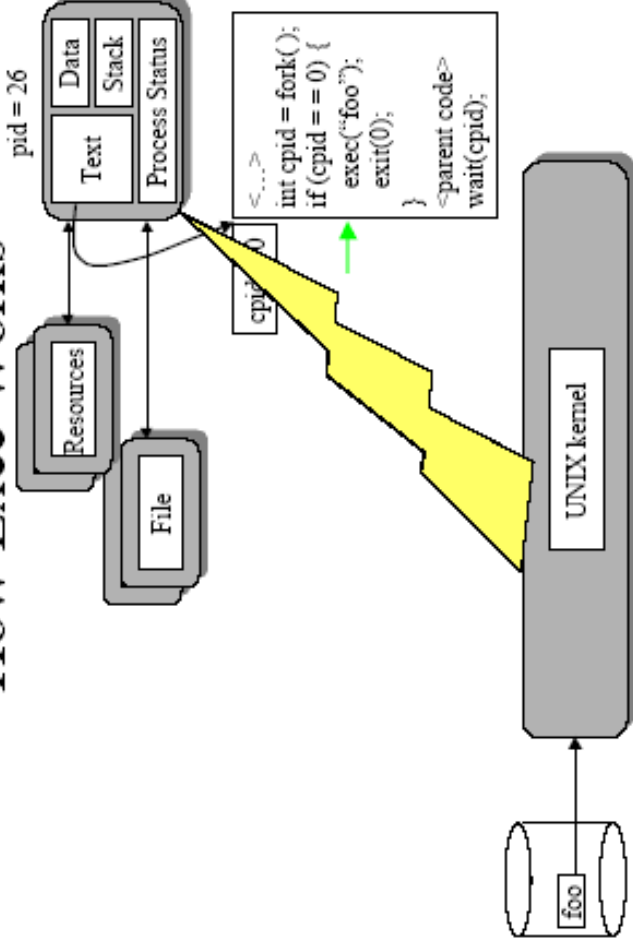


How Exec Works



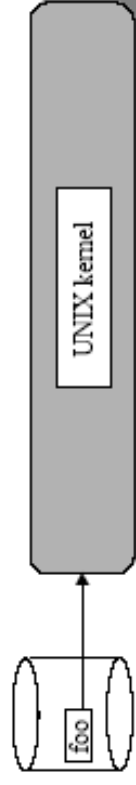
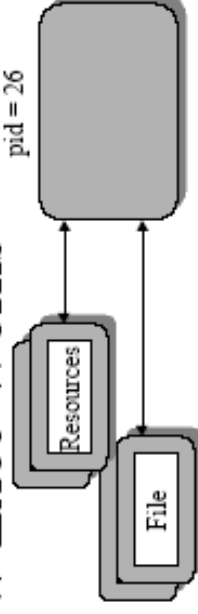


How Exec Works



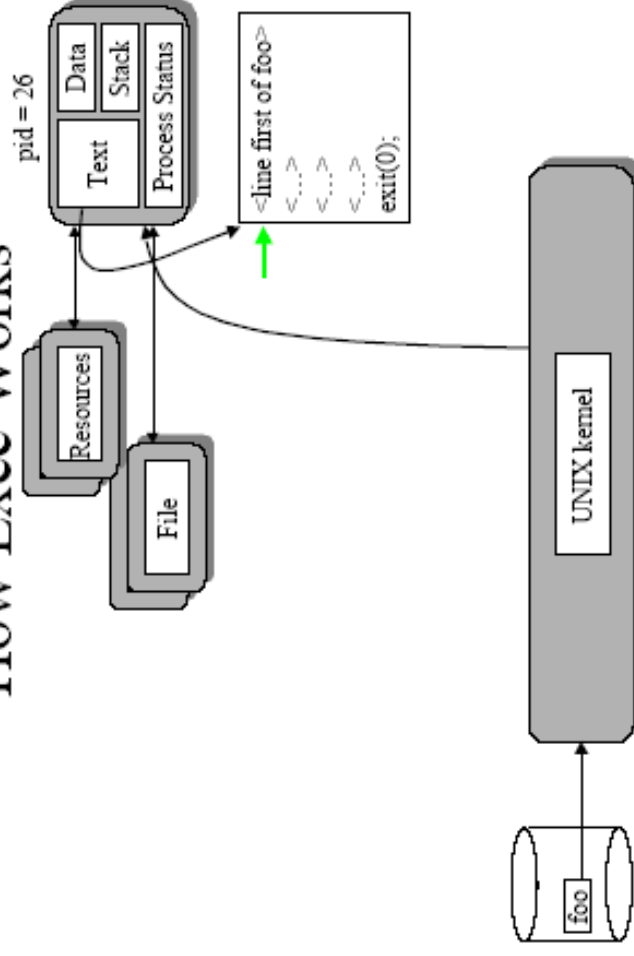


How Exec Works



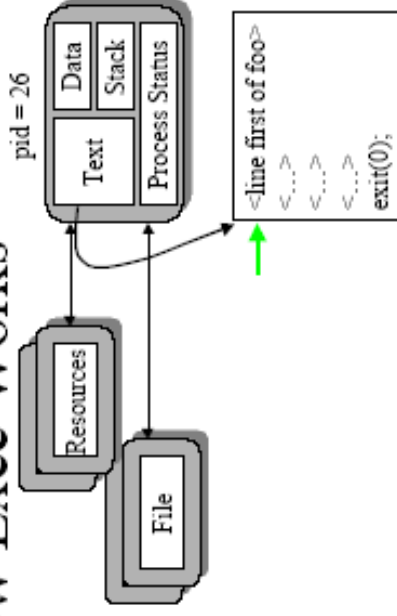


How Exec Works



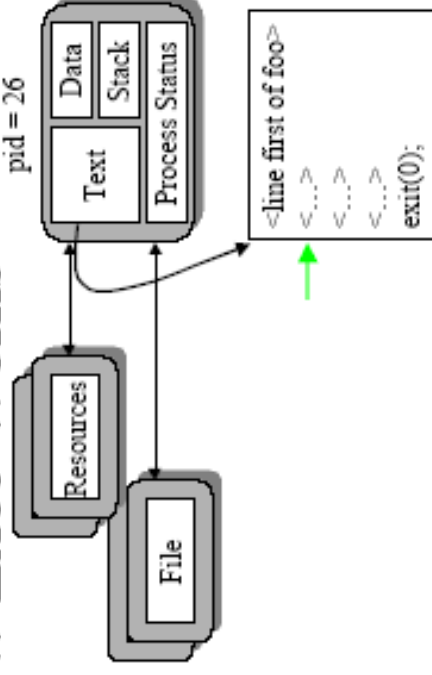


How Exec Works





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 родител - процесот е жив:
- Процесот – дете ја чува резултантната вредност додека таткото не ја побара, не умира туку влегува во `zombie/defunct` статус

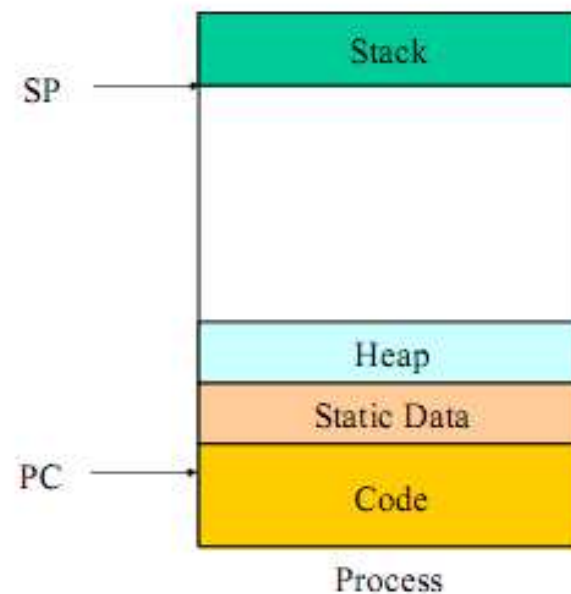


Карактеристики за процеси

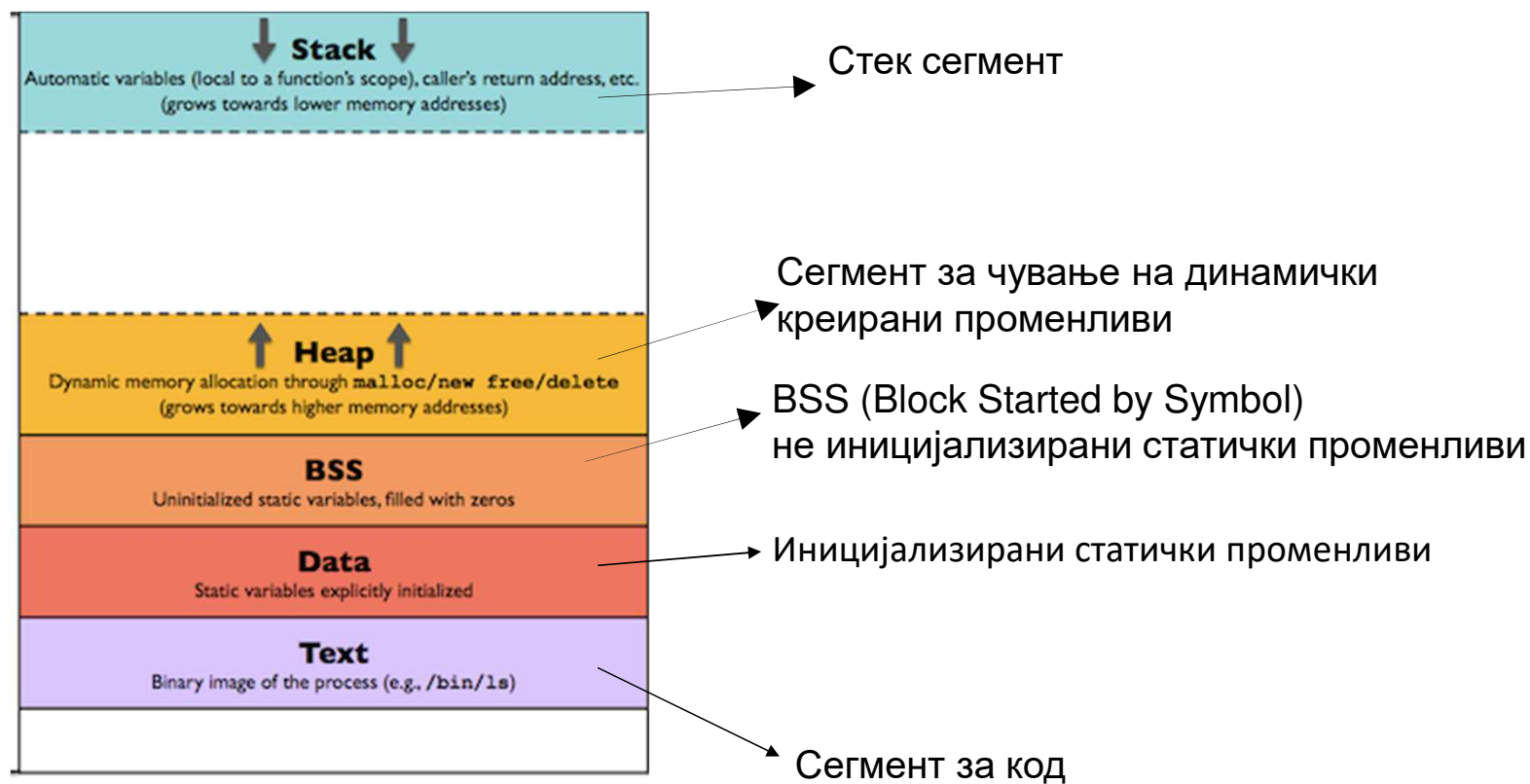
- Процесите имаат две карактеристики:
 - **Поседуваат ресурси**
 - **Тек на извршување** – следи тек (thread) на извршување
- Овие две карактеристики се третираат независно од ОС

Процес

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



Процес во меморија





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

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

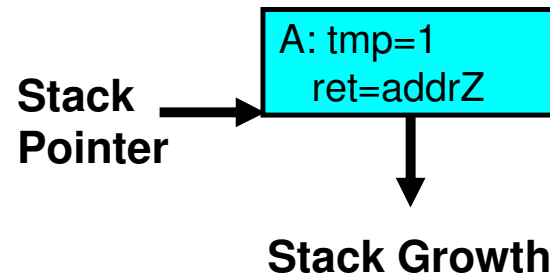
- ▶ Се чуваат привремени резултати
- ▶ Овозможува рекурзија
- ▶ Значајно за модерните јазици



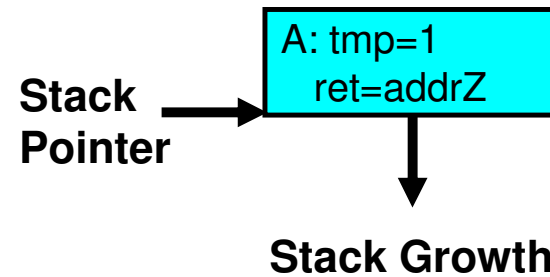
```

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

```



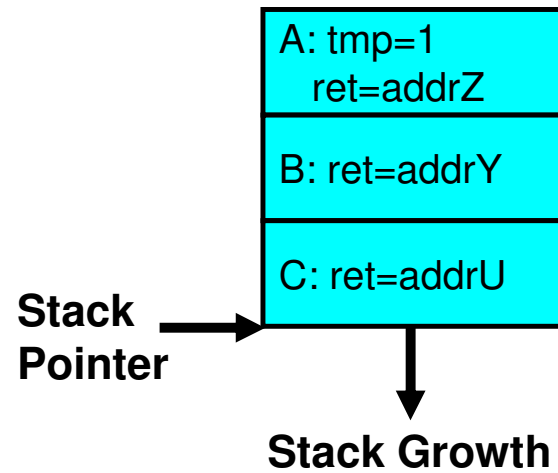
addrX:	A(int tmp) {
.	if (tmp<2)
.	B();
addrY:	printf(tmp);
.	}
.	B() {
.	C();
addrU:	}
.	C() {
.	A(2);
addrV:	}
.	A(1);
addrZ:	exit;



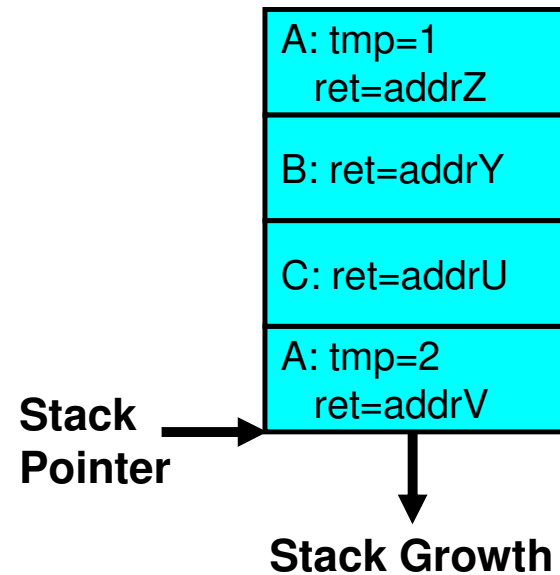

```

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

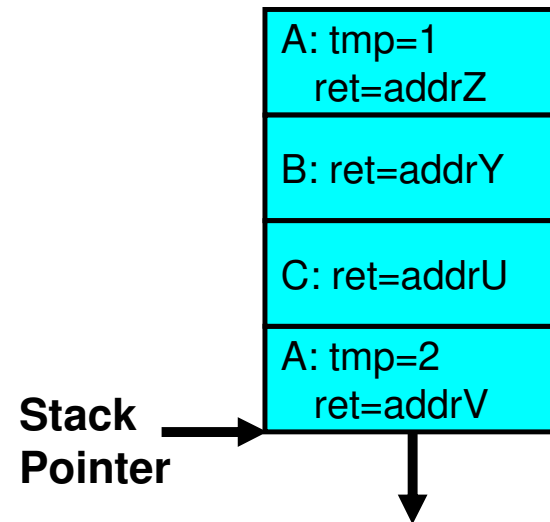
```



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;



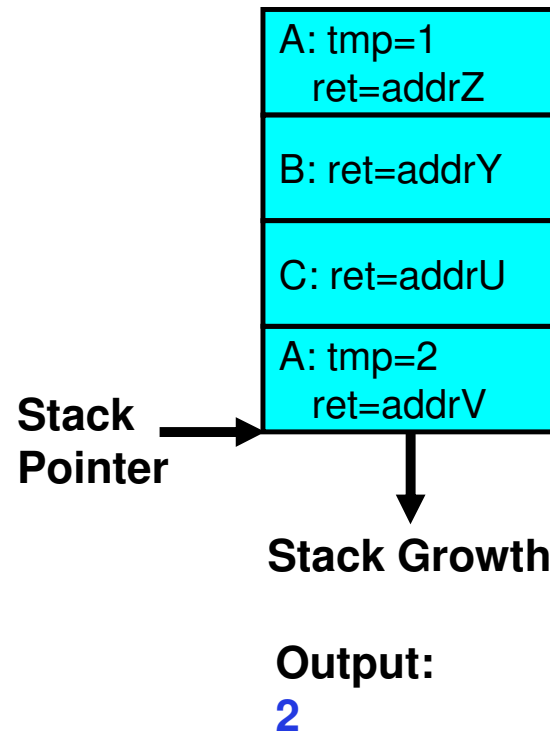
Output:



```

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

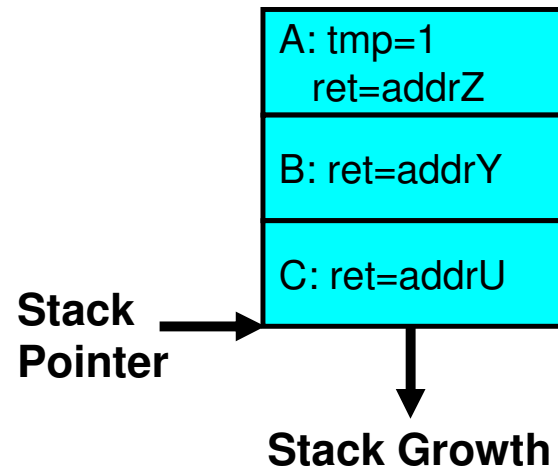
```



```

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

```



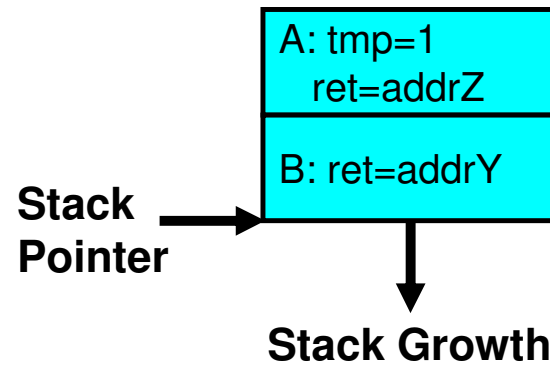
Output:
2



```

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) {
.
.
.
B();

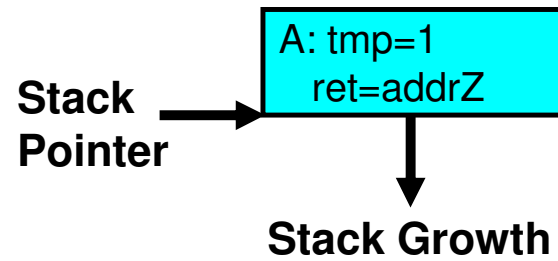
addrY: printf(tmp);

. }
.
.
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;

```

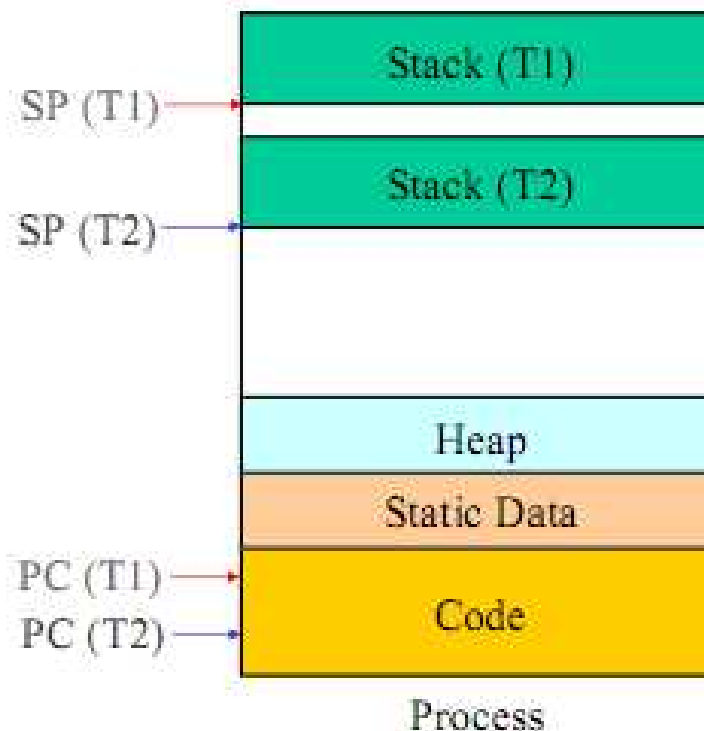
Output:

2
1

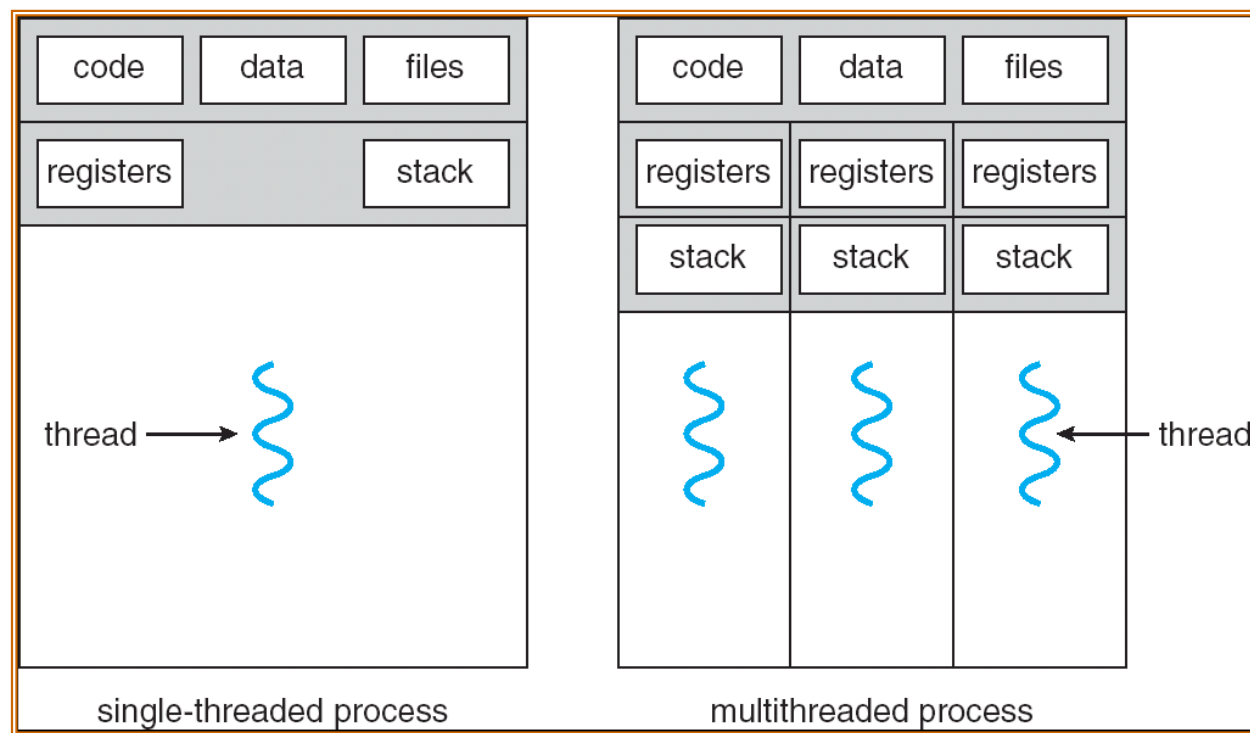


Процес со повеќе нишки

- 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

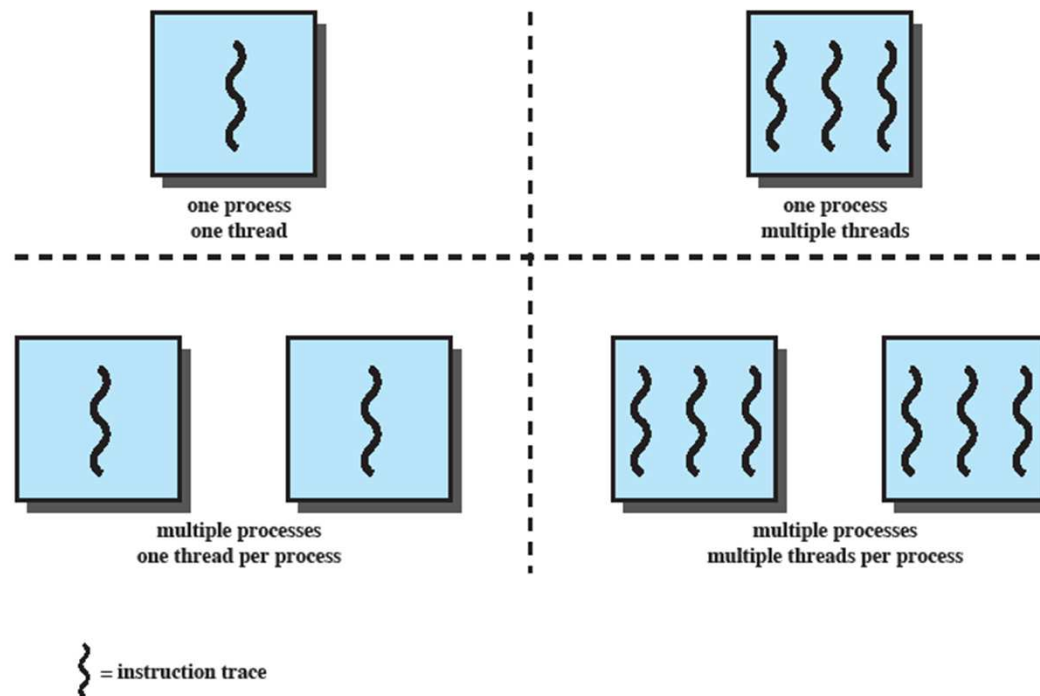


Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО



Повеќе нишки (Multithreading)

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



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

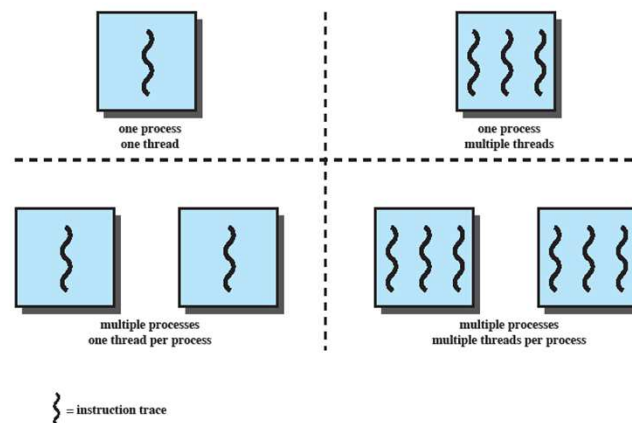


Figure 4.1 Threads and Processes [ANDE97]

Повеќе нишки

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

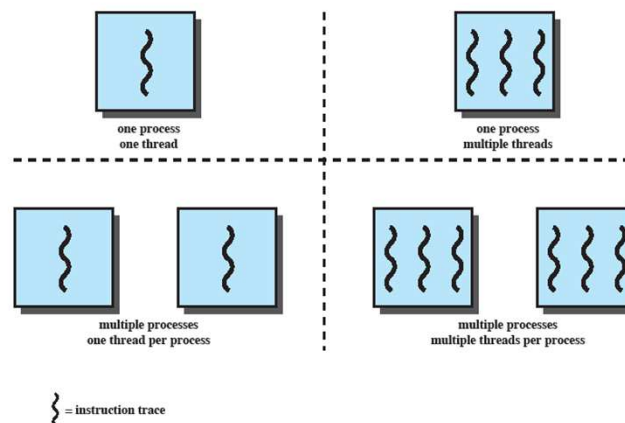


Figure 4.1 Threads and Processes [ANDE97]



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 нека во даден момент е потребно да се изврши промена на текстот
- Доколку има само еден процес (со една нишка), 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!!!

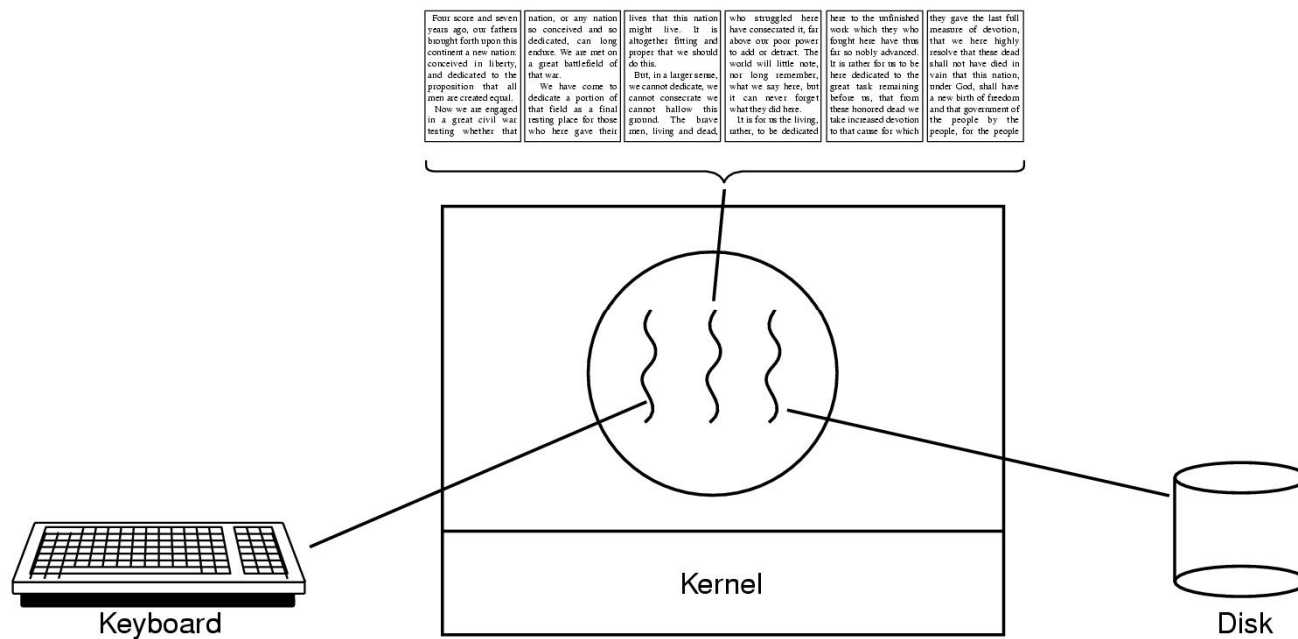


Пример 1 – пишување книга

- 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 нишки) не функционира во овој случај!



Пример - 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 нишката што повикува, а не целиот систем!



Thread vs. Process

- Нишките го споделуваат истиот адресен простор на процесот кој ги креирал
 - Процесите имаат сопствен независен адресен простор
- Нишките имаат директен пристап до податочниот сегмент на процесот
 - Процесите имаат своја копија на податочниот сегмент од родителот-процес
- Нишките можат директно да комуницираат помеѓу себе
 - Процесите мораат да користат посложена меѓу-процесна комуникација





Thread vs. Process

- Нишките немаат многу додатни податоци (overhead)
 - Процесите имаат значителна количина на додатни податоци
- Не постои посебен прекин за воведување на мултипрограмирање кај нишките, тие соработуваат
- Креирање на нишки е едноставно и брзо
 - Креирање на нов процес бара дуплирање на родител-процесот.
- Нишките можат да имаат контрола врз нишките од истиот процес
 - Процесите имаат контрола само врз процесите-деца.



Thread vs. Process

- Промени во главната нишка (откажување, промена на приоритет...) можат да имаат влијание на однесувањето на другите нишки
 - Промени кај родител-процесот не влијаат на децата-процеси.
- Сите нишки мораат да го извршуваат истиот извршен код
 - Децата-процеси можат да извршат комплетно различен код преку повик на `exec()`.
- За време на промена на нишка, виртуелниот мемориски простор останува ист
 - Кај промена на контекст на процес, се менува виртуелниот мемориски простор.



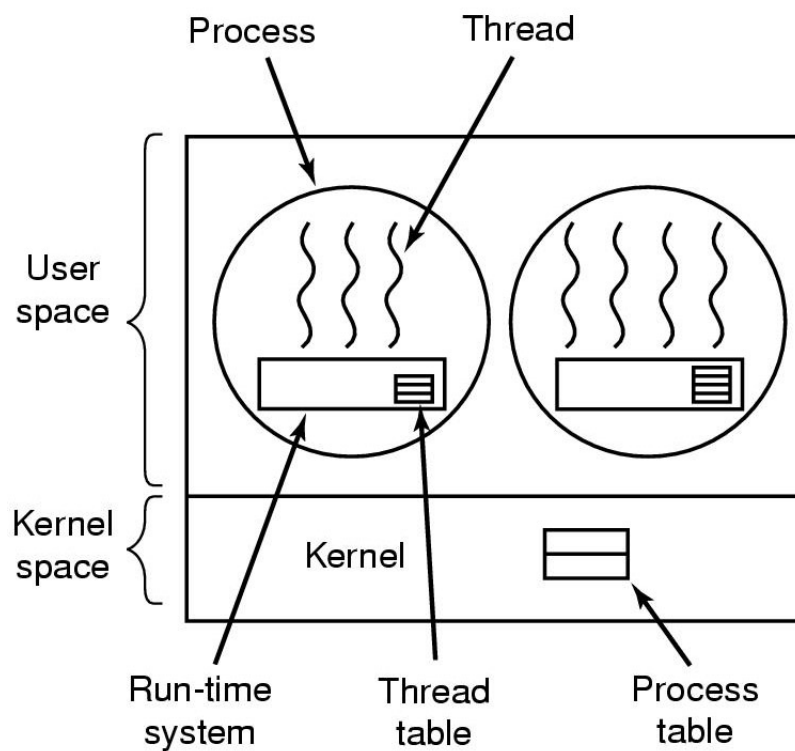
Thread vs. Process

- И процесите и нишките му ја препуштаат контролата на оперативниот систем при промена на контекст.
 - Процесот на предавање на контролата на јадрото па назад на процесите, заедно со промената на вредностите на регистрите се најголемата цена во изведување на промена на контекст.
 - И кај процесите и кај нишките ова време е најголемата цена која треба да се плати

Thread vs. Process

- При промена на контекст на процес, се пребришува TLB и кеш меморијата (на почетокот пристапот до меморијата е поскап)
 - Кај промена на контекст на нишки ова не се случува бидејќи нишките го споделуваат истиот адресен простор.
 - Често податоците во кеш меморијата од една нишка стануваат бескорисни за другата нишка (иако припаѓаат на ист процес)

Имплементација на нишки во корисничкиот простор

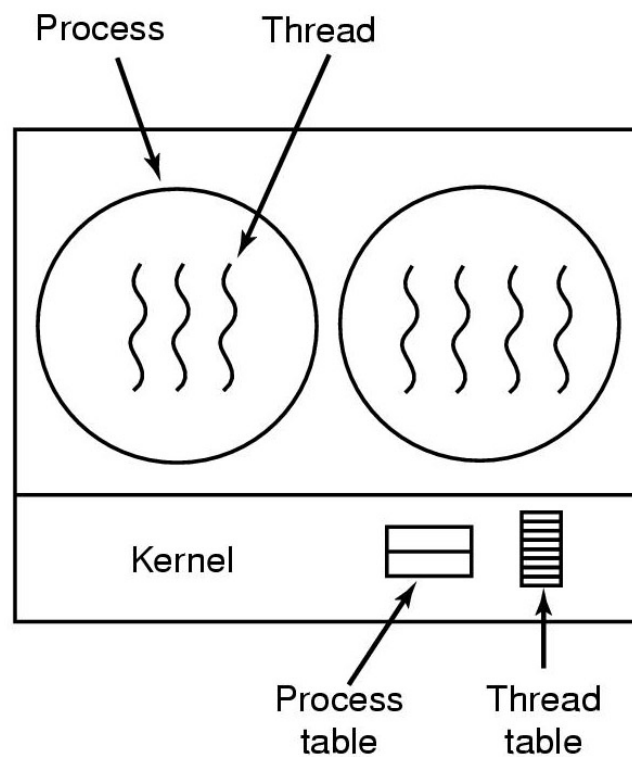




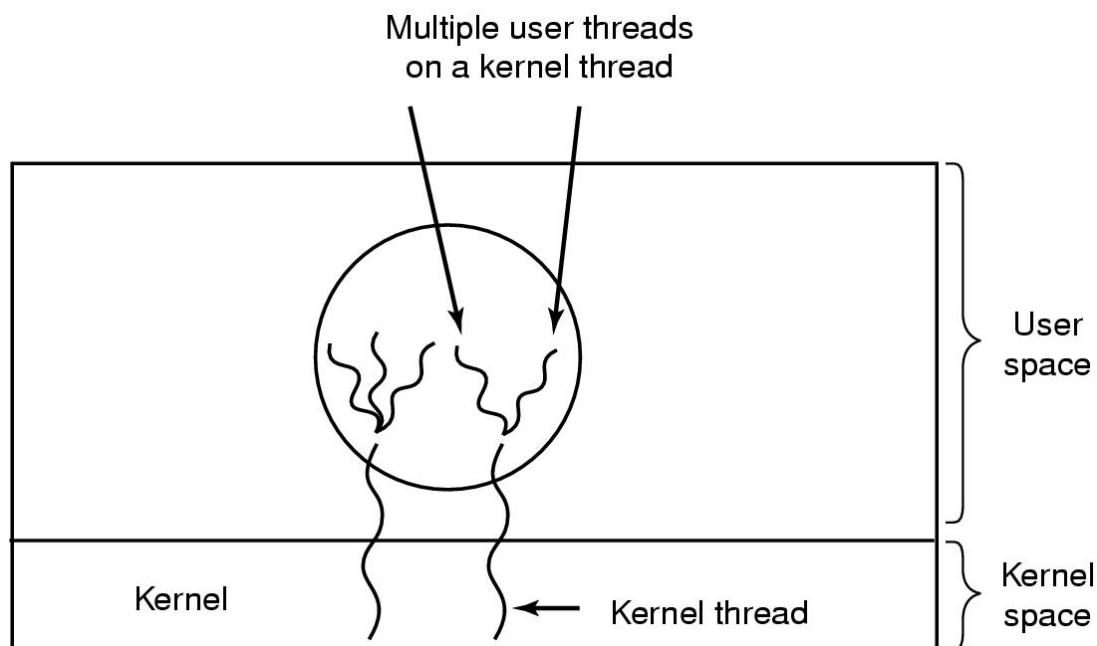
Имплементација на нишки во корисничкиот простор

- Предности
 - Добри перформанси
 - Секој процес може да има свој алгоритам за распределба
- Недостатоци
 - Проблеми со блокирачките системски повици
 - Проблеми со page fault
 - Проблеми со некооперативни нитки

Имплементација на нишки во јадрото на оперативниот систем



Хибридни имплементации



Комбинација на нишки во корисничкиот простор со оние во јадрото на оперативниот систем



Проблеми при миграција на Single-Threaded код во Multithreaded код

- Глобални променливи
- Истовремен повик на системски функции
- Обработка на прекините

