به نام ایزد یکتا

تمرین دوم درس سیستم عامل

استاد: دکتر جوادی

تهیه کننده: بردیا اردکانیان

۹۸۳۱۰۷۲

کد سوال‌ها به همراه عکس دیاگرام‌ها در فایل زیپ ضمیمه شده قابل مشاهده است

سوال اول)

در زبان برنامه نویسی C یک قانونی به اسم Lazy Evaluation به شرح ذیل وجود دارد:

- **||** will not evaluate the second expression if the first evaluates to a truthy value (not 0).
- **&&** will not evaluate the second expression if the first evaluates to a falsy value .(0)
- **&&** has greater precedence than **||**. So the "tricky" expression is equivalent to the fully parenthesized **`((fork() && fork()) || fork());`**

همچنین در خصوص fork می‌توان نوشت:

- fork returns 0 to the child process, and something other than 0 to the parent process. It returns -1 on failure, but evidently success is assumed in the problem.

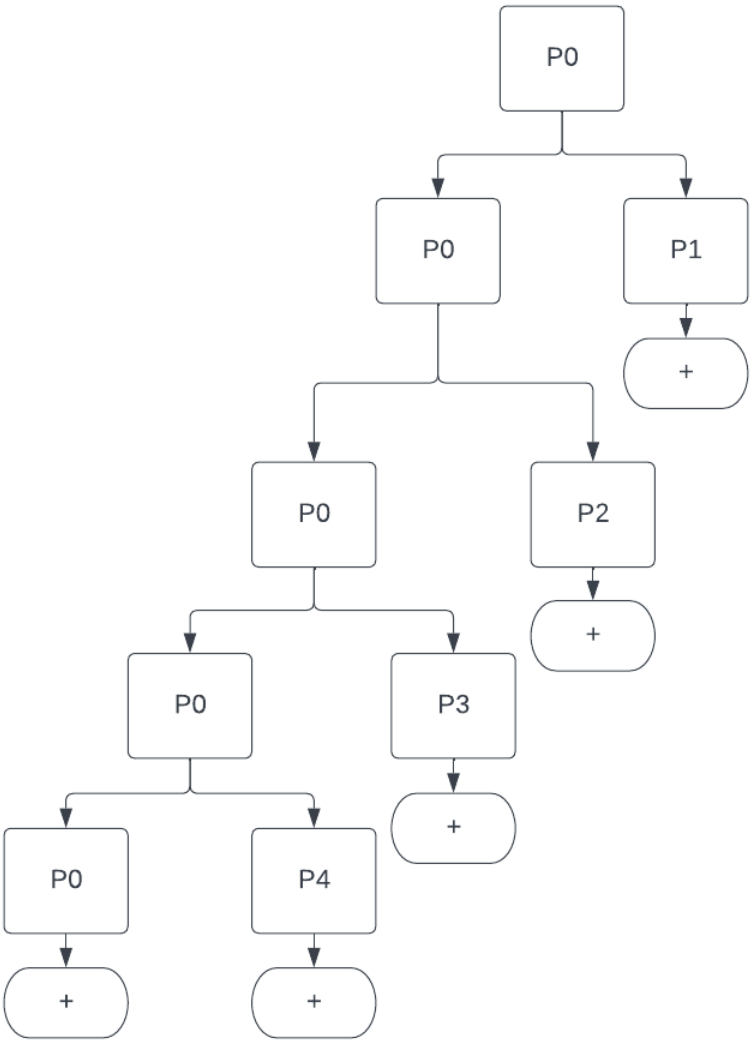
از این قوانین به موضوعات زیر پی می‌بریم:

1. اگر بعد از fork اوپراتون && وجود داشته باشد، با علم به این‌که خروجی فرزند صفر و خروجی پدر غیر صفر است؛ در فرآیند پدر به خواندن ادامه expression می‌پردازیم ولی در فرآیند فرزند دیگر ادامه expression را نخوانده و به خط بعدی می‌رویم
2. اگر بعد از fork اوپراتون || وجود داشته باشد، با علم به این‌که خروجی فرزند صفر و خروجی پدر غیر صفر است؛ در فرآیند فرزند به خواندن ادامه expression می‌پردازیم ولی در فرآیند پدر دیگر ادامه expression را نخوانده و به خط بعدی می‌رویم
3. ترکیب موارد پیش نیز ممکن است که اولویت با && و بعد با || می‌باشد

در تمامی شکل‌ها فرآیند فرزند در سمت راست و فرآیند والد در سمت چپ است

الف)

با توجه به مطالب ذکر شده، با هر بار صدا زده شدن fork، فرآیند فرزند + چاپ می‌کند و فرآیند پدر ادامه برنامه را اجرا می‌کند تا در نهایت + را چاپ کند. در نتیجه پنج عدد + چاپ می‌شود.



عکس 1-1

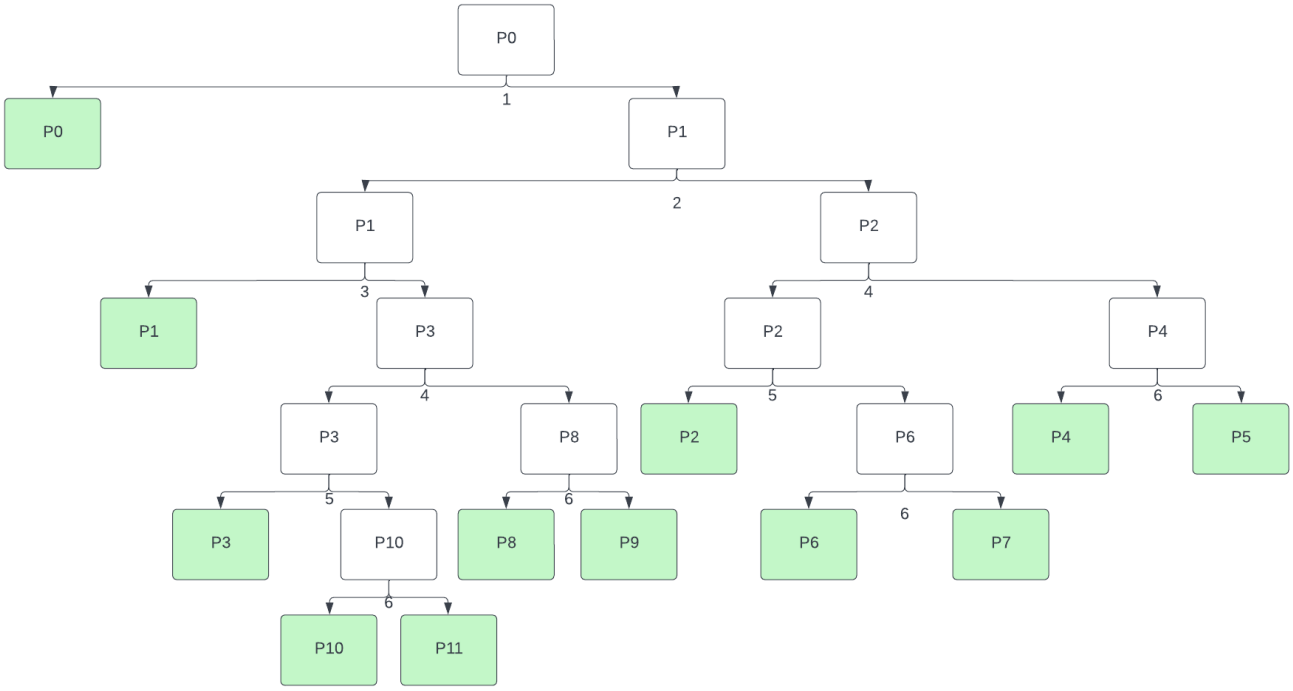
ب)

در این برنامه با فراخوانی fork اگر در فرآیند پدر باشیم و اوپراتور بعدی || باشد، برنامه به خطی بعدی (چاپ +) می‌رود و اگر در فرآیند فرزند باشیم ادامه expression را می‌خواند.

اگر در فرآیند پدر باشیم و اوپراتور بعدی && باشد، ادامه expression را می‌خواند و اگر در فرآیند فرزند باشیم برنامه به خطی بعدی (چاپ +) می‌رود.

اگر به هر fork را (از چپ به راست) شماره‌های یک تا شش را اختصاص دهیم درختوار فرآیندها به صورت شکل 1-2 نمایش داده می‌شود.

در نتیجه 12 عدد + چاپ می‌شود. ترتیب حدودی چاپ آنها با شماره فرآیند مشخص شده است.



عکس 1-2

سوال دوم)

در حالت کلی هر فرآیند (process) توسط مشخصه فرآیندش شناخته می‌شود که یک عدد صحیح منحصر بفرد است. یک فرآیند جدید با فراحوانی سیستمی fork ایجاد می‌شود. فرآیند جدید شمامل یک گپی از فضای آدرس فرآیند است. این مکانیزم به فرآیند والد اجازه می‌دهد که به آسانی با فرآیند فرزند ارتباط برقرار کند. هر دو فرآیند، پس از دستورالعمل fork(1) ادامه می‌یابند، اما با یک تفاوت: کد برگشت فراخوانی fork برای فرآیند فرزند برابر با صفر ولی برای فرآیند والد غیر صفر است.

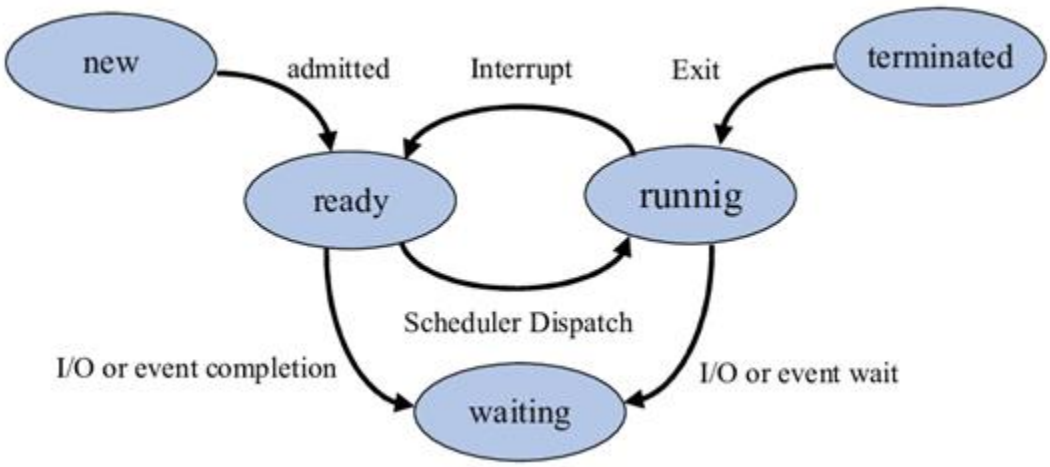
فراخوانی سیستمی exec، پس از فراخوانی سیستمی fork توسط یکی از این دو فرآیند استفاده می‌شود که توسط آن، فضای آدرس فرآیندها با یک برنام جدید جایگزین می‌گردد. فراخوانی سیستمی exec، یک فایل باینری را بدون حافظه، بار کرده و اجرای آن را آغاز می‌کند. در این زمان، دو فرآیند قادرند که با هم ارتباط برقرار کنند. ما هم اکنون دو فرآیند متفاوت داریم که بر روی یک کپی از یک برنامه اجرا می‌شوند.

مقدار pid برای فرآیند فرزند صفر و برای فرآیند والد مقداری بزرگ تر از صفر است.

سوال سوم)

```c
#include<stdlib.h>

int main()
{
	int i = 1;
	while (i < 100) i++;
	printf("%d", i);
	while (i > 0) i--;
	printf("%d", i);

	return 0;
}
```

چهار نوع process state داریم New، Ready، Running، Waiting، و Terminated.



عکس 3-1

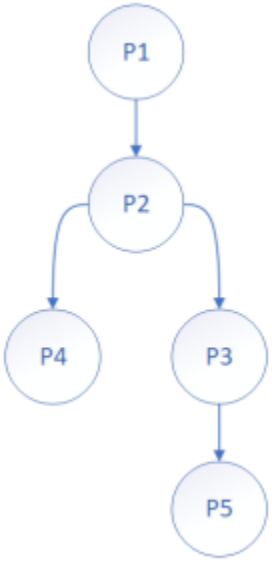
در ابتدا فرآیند در وضعیت New می‌باشد و ایجاد می‌شود. در مرحله بعدی به وضعیت Ready می‌رود و منتظر می‌ماند تا پردازش شود. چون تنها فرآیند سیستم است وارد وضعیت Running می‌شود و خط به خط برنامه را اجرا می‌کند.

1. متغیر i اینجاد و مقداردهی می‌شود
2. حلقه به طور کامل انجام می‌شود و در نهایت مقدار i برابر با 100 می‌شود
3. با رسیدن به دستور printf مقدار i را به تابع می‌فرستد و از وضعیت Running به وضعیت Waiting می‌رود تا درخواست I/O (نمایش مقدار i) انجام شود

4. با چاپ شدن مقدار i، فرآیند دوباره به وضعیت Ready می‌رود
5. چون تنها فرآیند سیستم است به وضعیت Running تغییر وضعیت می‌دهد و به اجرای ادامه برنامه می‌پردازد
6. حلقه دوم کامل اجرا می‌شود و مقدار i صفر می‌شود
7. با رسیدن به دستور printf مقدار i را به تابع می‌فرستد و از وضعیت Running به وضعیت Waiting می‌رود تا درخواست I/O (نمایش مقدار i) انجام شود.
8. با چاپ شدن مقدار i، فرآیند دوباره به وضعیت Ready می‌رود
9. چون تنها فرآیند سیستم است به وضعیت Running تغییر وضعیت می‌دهد
10. با توجه به اینکه برنامه به پایان رسیده است، از وضعیت Running به Terminated تغییر وضعیت می‌دهد

سوال چهارم)

برای به وجود آوردن درخت فرآیند زیر از سه برنامه استفاده می‌کنیم که یکدیگر را صدا می‌کنند.



شکل 4-1

برنامه اول (اصلی):

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

int main()
{
  // p1
  // pid
  printf("Main program with pid=%d\n", getpid());

  // init program args and execv
  char *args[] = {"search-sort", "c", "program", NULL};
  execv("./search-sort", args); // p2

  printf("Back in main program pid=%d\n", getpid())
}
```

برنامه دوم:

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

int main()
{
  // p2
  // pid
  printf("Search-Sort program with pid=%d\n", getpid());

  // fork
  // p3 (sort) algorithm
  pid_t pid;
  pid = fork();
  pid = fork();
  // p4 (search) algorithm
  pid_t pid2;
  pid2 = fork()

  if (pid2 < 0) { // fork failed p3
    printf("Fork 1 failed\n");
  }
  if (pid < 0) { // fork failed p4
    printf("Fork 2 failed\n");
  }

  // p3 decision tree
  if (pid == 0) { // child process // p3
    printf("Entering sort program with pid=%d\n", getpid());
    // sort section
    // sort program args and execv
    char *args[] = {"sort", "c", "program", NULL};
    execv("./sort", args); // p5
  } else if (pid > 0) { // parent process // p2
    printf("Parent process with pid=%d\n", getpid());
    waitpid(pid, &returnStatus, 0); // wait for sort section to be completed (p3)
    // p3 is finished. kill p4
    kill(pid, SIGKILL);
    printf("Child completed"); // child (p3) process finished
  }

  // p4 decision tree
  if (pid2 == 0) { // child process // p4
    // p4
    // pid
```

```
    // search program
    printf("Search section with pid=%d\n", getpid());
    search();
    printf("Search section finished pid=%d\n", getpid());

  } else if (pid2 > 0) { // parent process // p2
    printf("Parent process with pid=%d\n", getpid());
    waitpid(pid2, &returnStatus, 0); // waits for p4 to be completed
    printf("Child completed")
  }

  printf("Search-Sort program finished with pid=%d\n", getpid());
  printf("Terminate\n");
}
```

برنامه سوم (Sort):

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

int main()
{
  // p5
  // pid
  // sort program
  printf("Sort program with pid=%d\n", getpid());
  sort();
  printf("Sort program finished pid=%d\n", getpid());
}
```

توضیح کد:

- برنامه اول (فرایند P1) با کمک exec برنامه دوم (فرآیند P2) را فراخوانی و اجرا می‌کند
- برنامه دوم با کمک fork فرزند اول (فرایند P3) و والد اول (فرایند P2) را می‌سازد
- برنامه دوم با کمک fork فرزند دوم (فرایند P4) و والد دوم (فرایند P2) را می‌سازد
- والد اول و دوم فرآیند یکسانی هستند صرفا برای توضیحات اول و دوم را نوشته‌ایم
  - برنامه فرزند اول
    - برنامه فرزند اول برنامه سوم (فرآیند P5) را با کمک exec اجرا می‌کند (برنامه قسمت Sort (شرط اول)

- با اتمام فرآیند فرزند اول وقفه‌ای برای فرآیند والد اول فرستاده می‌شود تا از wait خارج شود
  - برنامه والد اول (فرآیند P2)
    - برنامه والد با کمک wait منتظر به اتمام رسیدن برنامه فرزند اول می‌ماند
    - با به اتمام رسیدن برنامه فرزند اول (فرآیند P3)، فرایند فرزند دوم (P4) را از بین می‌برد (P2 منتظر خاتمه فرآیند P3 می‌ماند تا P4 را از بین ببرد) (شرط سوم)
  - فرزند دوم (فرآیند P4)
    - فرزند دوم برنامه Search را اجرا می‌کند (شرط دوم)
    - با اتمام فرآیند فرزند دوم وقفه‌ای برای فرآیند والد دوم فرستاده می‌شود تا از wait خارج شود
  - برنامه والد دوم (فرآیند P2)
    - برنامه والد با کمک wait منتظر به اتمام رسیدن برنامه فرزند دوم می‌ماند
- با به اتمام رسیدن فرایند والد اول و دوم، برنامه دوم به پایان می‌رسد (پردازه P2 از بین می‌رود)
- با به پایان رسیدن برنامه دوم، برنامه اول نیز به پایان می‌رسد (پردازه P1 از بین می‌رود)

سوال پنجم)

از معلومات سوال اول استفاده می‌کنیم:

در زبان برنامه نویسی C یک قانونی به اسم Lazy Evaluation به شرح ذیل وجود دارد:

- || will not evaluate the second expression if the first evaluates to a truthy value (not 0).
- && will not evaluate the second expression if the first evaluates to a falsy value .(0)
- && has greater precedence than ||. So the "tricky" expression is equivalent to the fully parenthesized `((fork() && fork()) || fork());`

همچنین در خصوص fork می‌توان نوشت:

- fork returns 0 to the child process, and something other than 0 to the parent process. It returns -1 on failure, but evidently success is assumed in the problem.

از این قوانین به موضوعات زیر پی می‌بریم:

1. اگر بعد از fork اوپراتون && وجود داشته باشد، با علم به این‌که خروجی فرزند صفر و خروجی پدر غیر صفر است؛ در فرآیند پدر به خواندن ادامه expression می‌پردازیم ولی در فرآیند فرزند دیگر ادامه expression را نخوانده و به خط بعدی می‌رویم
2. اگر بعد از fork اوپراتون || وجود داشته باشد، با علم به این‌که خروجی فرزند صفر و خروجی پدر غیر صفر است؛ در فرآیند فرزند به خواندن ادامه expression می‌پردازیم ولی در فرآیند پدر دیگر ادامه expression را نخوانده و به خط بعدی می‌رویم
3. ترکیب موارد پیش نیز ممکن است که اولویت با && و بعد با || می‌باشد

در تمامی شکل‌ها فرآیند فرزند در سمت راست و فرآیند والد در سمت چپ است

الف)

```
int main()
{
  if(fork() && !(fork())) {
    if(fork() || fork())
      fork();
  }
  printf("2");
```

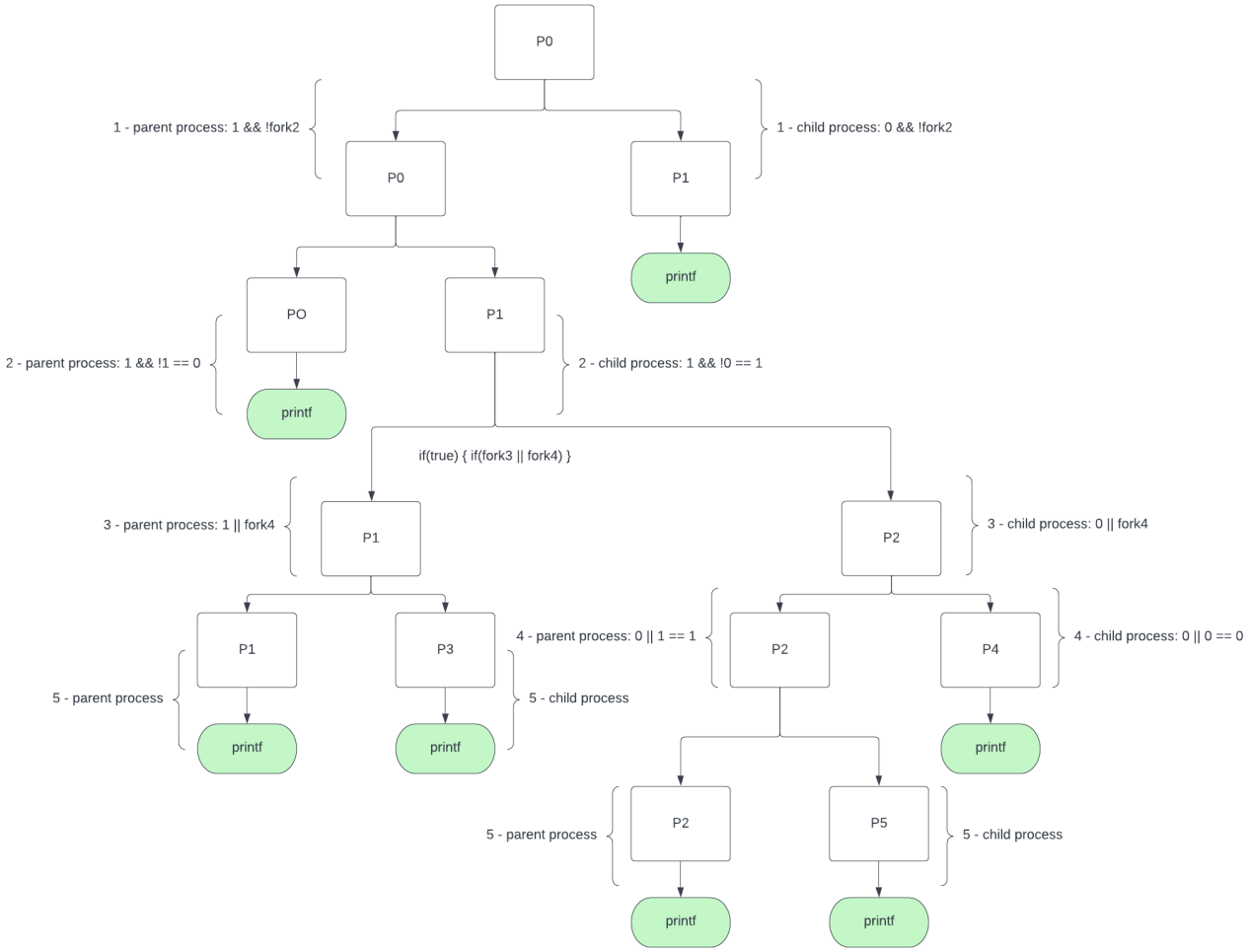
```
  return 0;
}
```

- در این برنامه با فراخوانی fork اگر در فرآیند پدر باشیم و اوپراتور بعدی || باشد، برنامه ادامه expression را نمی‌خواند می‌رود و اگر در فرآیند فرزند باشیم ادامه expression را می‌خواند.
- اگر در فرآیند پدر باشیم و اوپراتور بعدی && باشد، ادامه expression را می‌خواند و اگر در فرآیند فرزند باشیم برنامه ادامه expression را نمی‌خواند.

به همین ترتیب برنامه می‌بایست هفت بار عدد دو را چاپ کند. تحلیل فرآیندها در عکس 5-1 قابل مشاهده است.

شماره گذاری forkها به شرح ذیل است:

```
int main()
{
  if(1 && 2) {
    if(3 || 4)
      5;
  }
  printf("2");
  return 0;
}
```


P0
1 - parent process: 1 && !fork2
1 - child process: 0 && !fork2
P0
P1
printf
PO
P1
2 - parent process: 1 && !1 == 0
2 - child process: 1 && !0 == 1
printf
if(true) { if(fork3 || fork4) }
3 - parent process: 1 || fork4
P1
P2
3 - child process: 0 || fork4
P1
P3
4 - parent process: 0 || 1 == 1
P2
P4
4 - child process: 0 || 0 == 0
5 - parent process
5 - child process
printf
printf
printf
5 - parent process
P2
P5
5 - child process
printf
printf

عکس 5-1

ب)

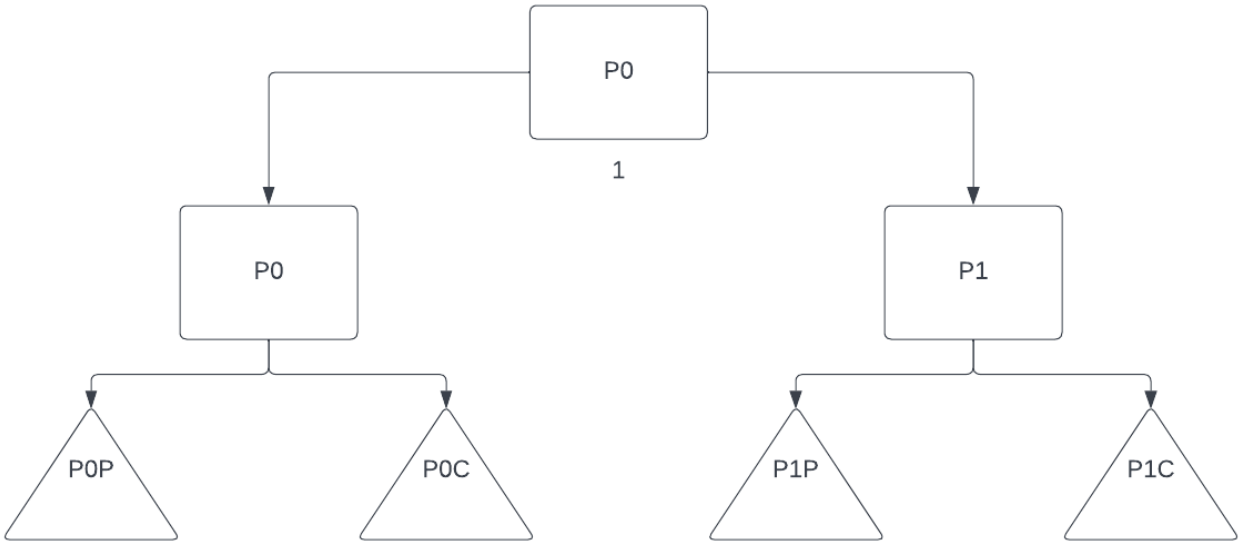
در این برنامه نیز مانند قسمت الف تحلیل می‌کنیم.

```
int main()
{
  fork();
  fork() && fork() || fork();
  fork();

  printf("hello\n");
  return 0;
}
```

شماره گذاری forkها به شرح ذیل است:

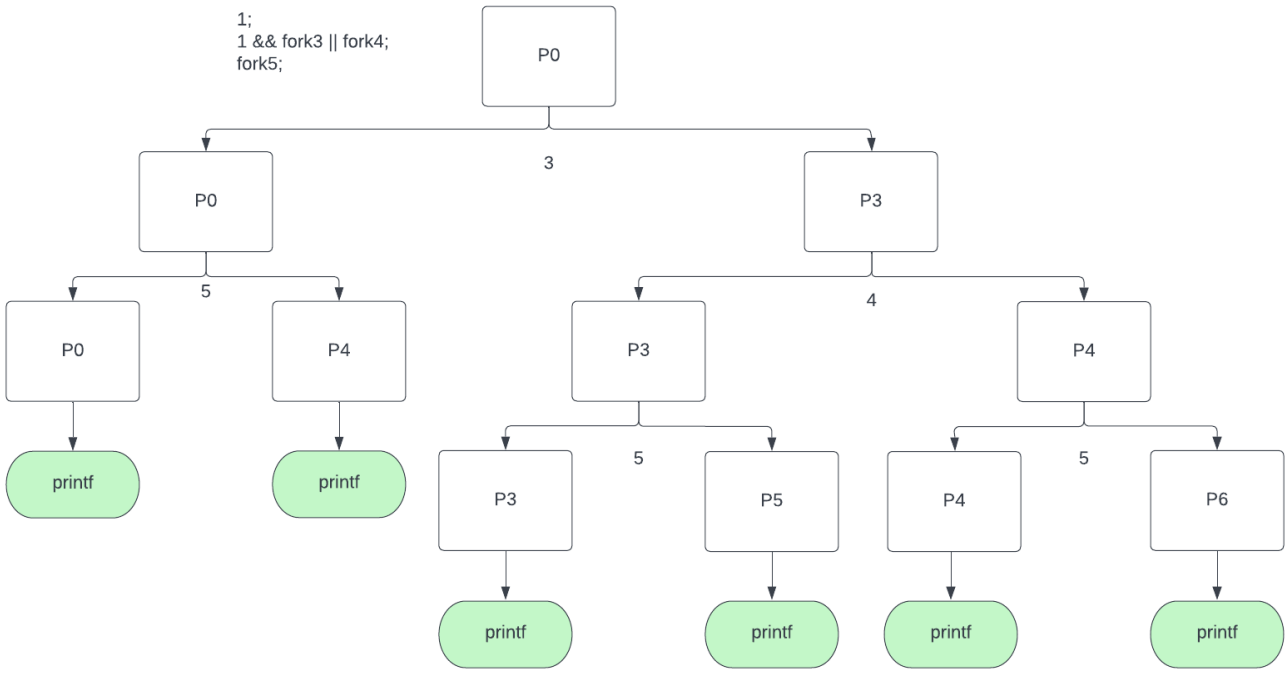
```
int main()
{
  1;
  2 && 3 || 4;
  5;

  printf("hello\n");
  return 0;
}
```

به همین ترتیب برنامه می‌بایست ۲۰ بار عبارت «hello» را چاپ کند. تحلیل فرآیندها در عکس‌های ذیل قابل مشاهده است.



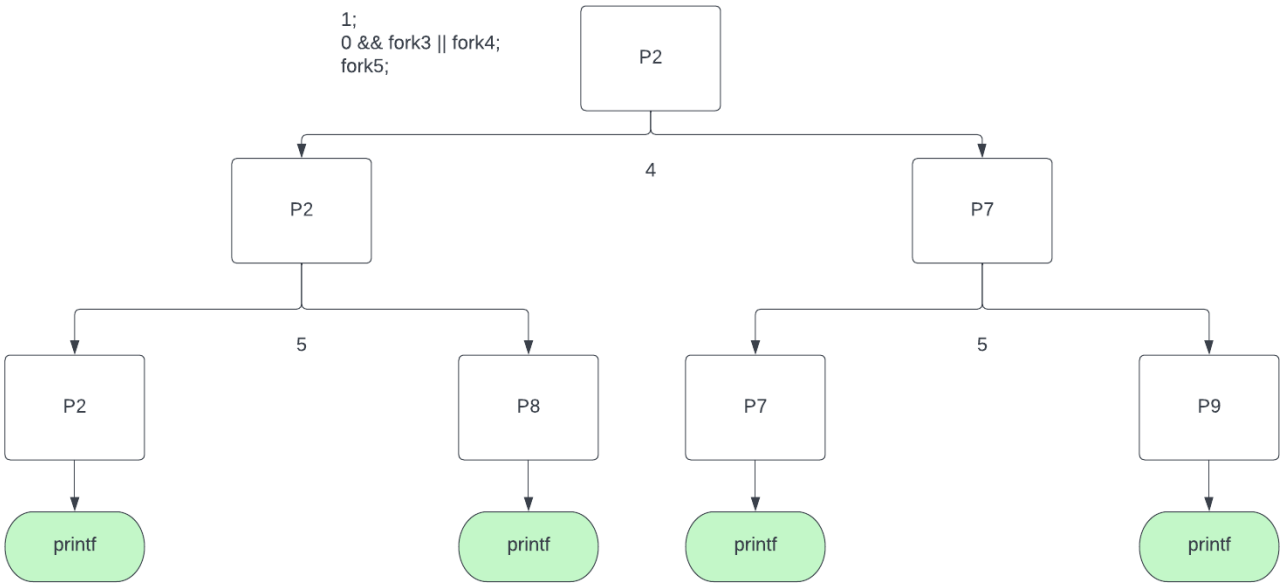
عکس 5-2-1

دیاگرام خام



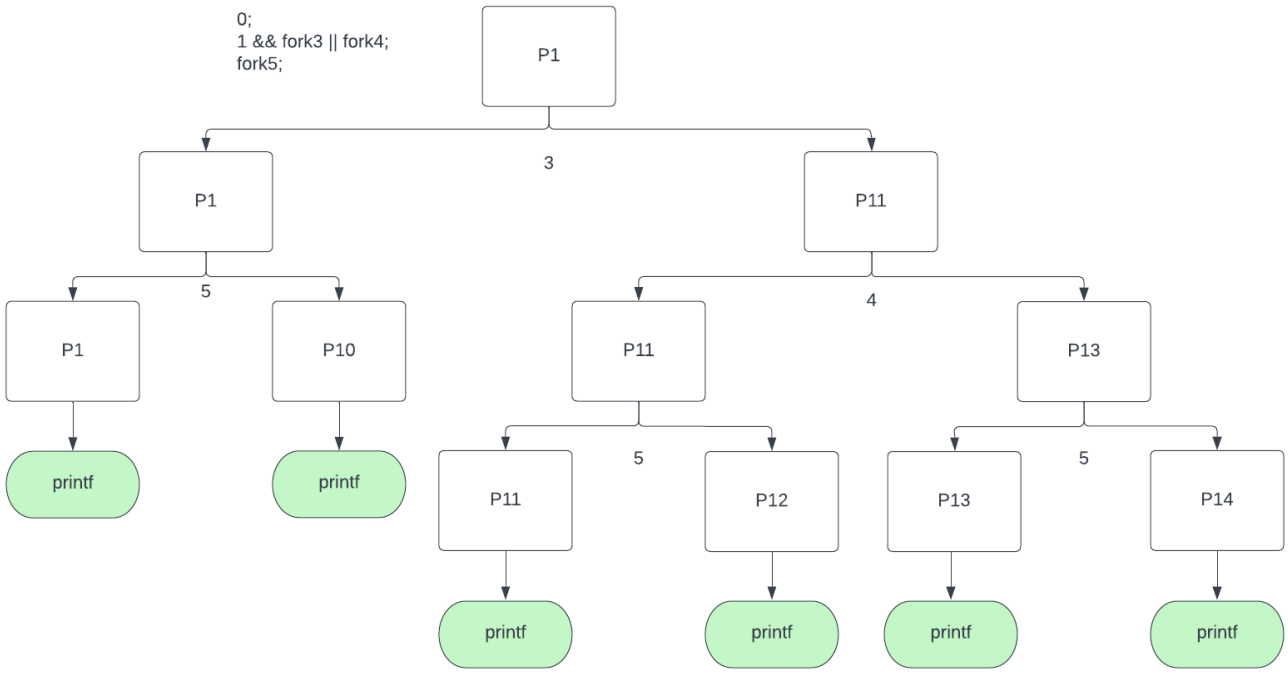
عکس 5-2-2

زیردرخت P0P (P0 Parent)



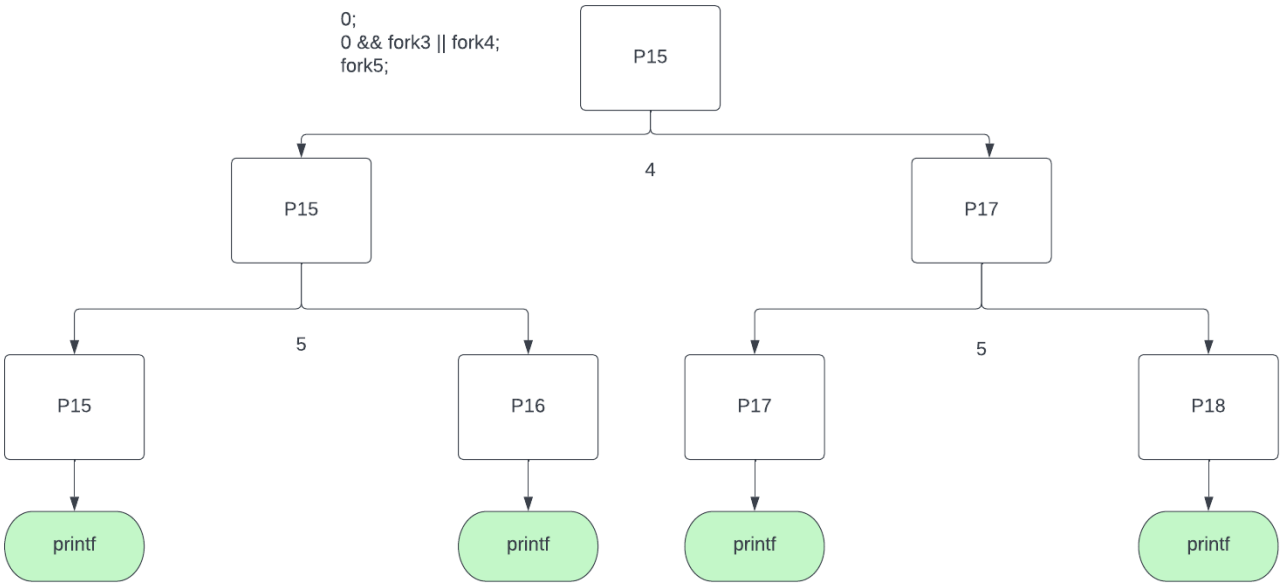
عکس 5-2-3

زیردرخت P0C (P0 Child)



عکس 5-2-4

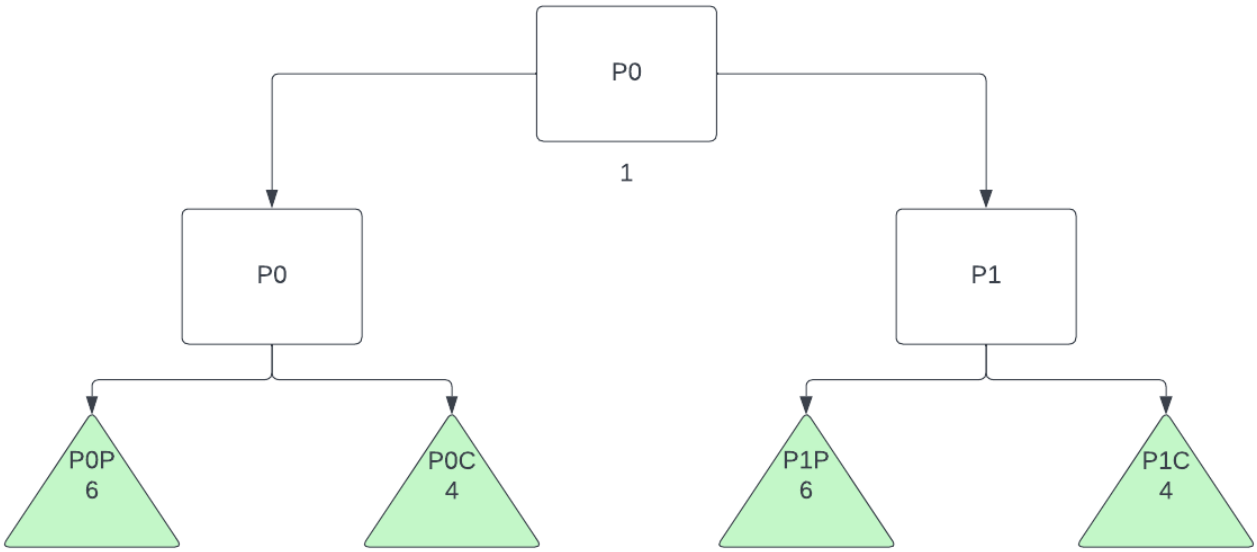
زیردرخت P1P (P1 Parent)



عکس 5-2-5

زیردرخت P1C (P1 Child)

در نتیجه هر زیردرخت P0 (متشکل از ریزدرخت‌های P0P و P0C) و P1 (متشکل از ریزدرخت‌های P1P و P1C) ۱۰ بار عبارت مربوطه را چاپ می‌کنند. تعداد کل دفعات چاپ عبارت «hello» ۲۰ بار می‌باشد.



عکس 1-2-5

دیاگرام کامل