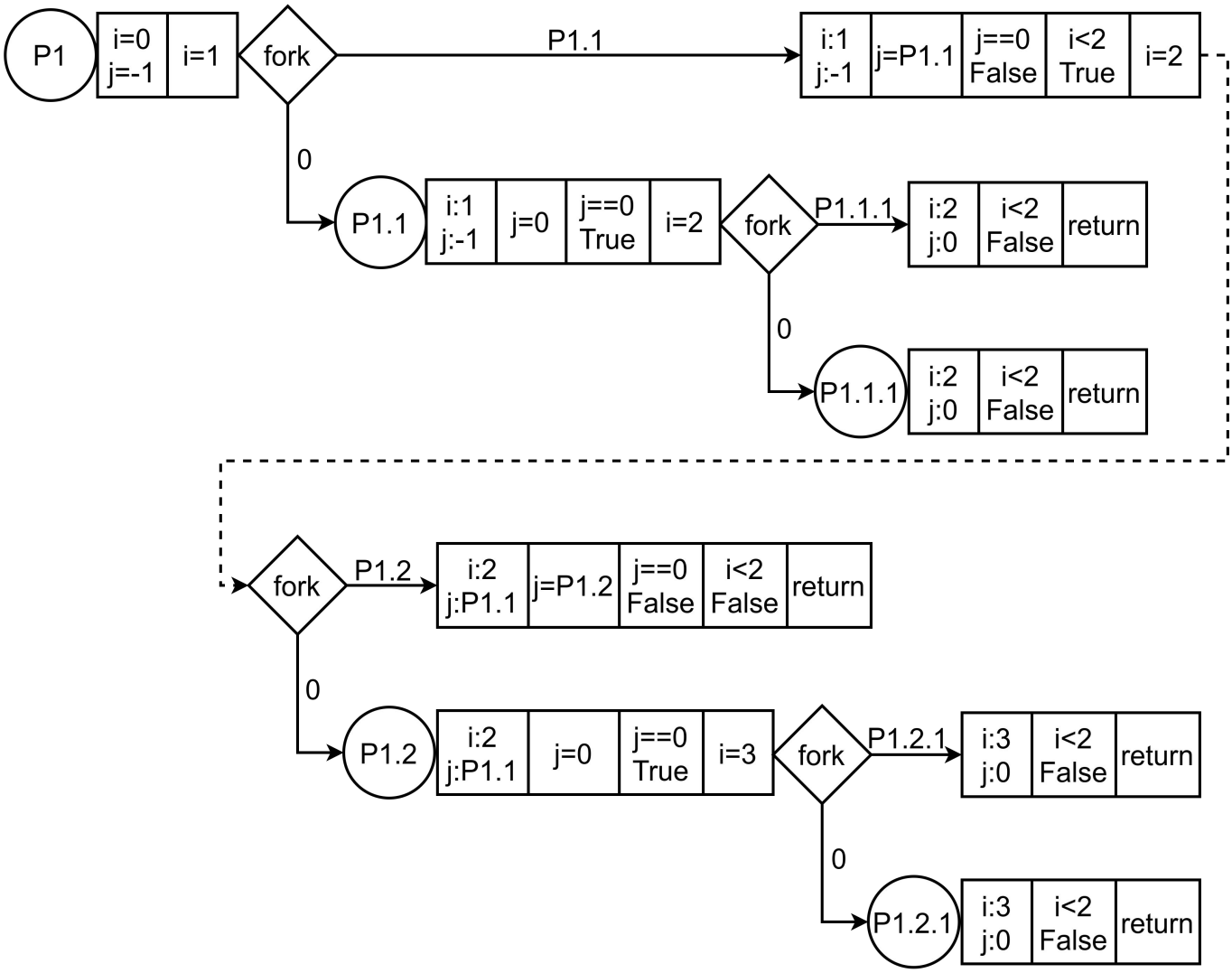
```
// 1.1.a

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
 int i=0;

 fork();

 while (1) {
        printf("pid:%d i:%d\n", getpid(), i++);
 }
}
```

```
// 1.1.b
// #include directives equal to those of program 1.2.a.c

main()
{
 int i=0;

 fork();

 while (1) {
        printf("pid:%d i:%d\n", getpid(), i++);
        sleep(1);
 }
}
```

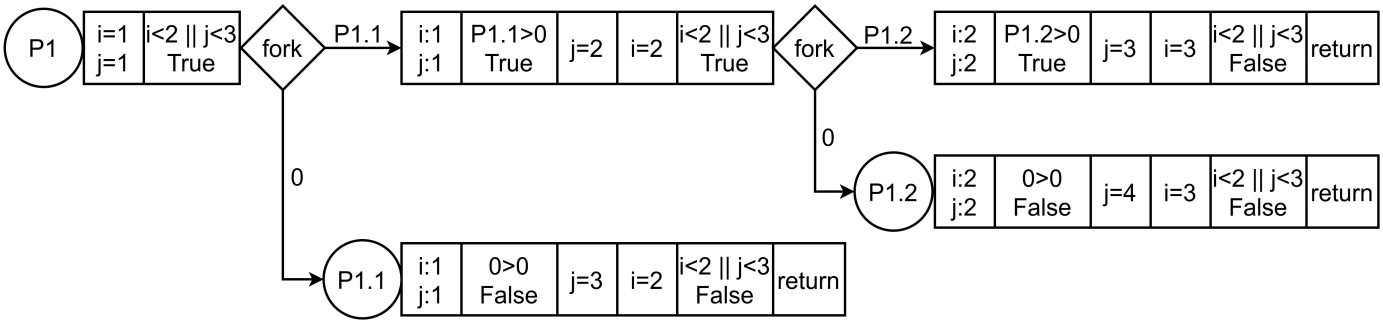
```
// 1.1.c
#include <unistd.h>

main()
{
 fork();

 while (1);
}
```

```
// 1.1.d

main()
{
 while (1);
}
```

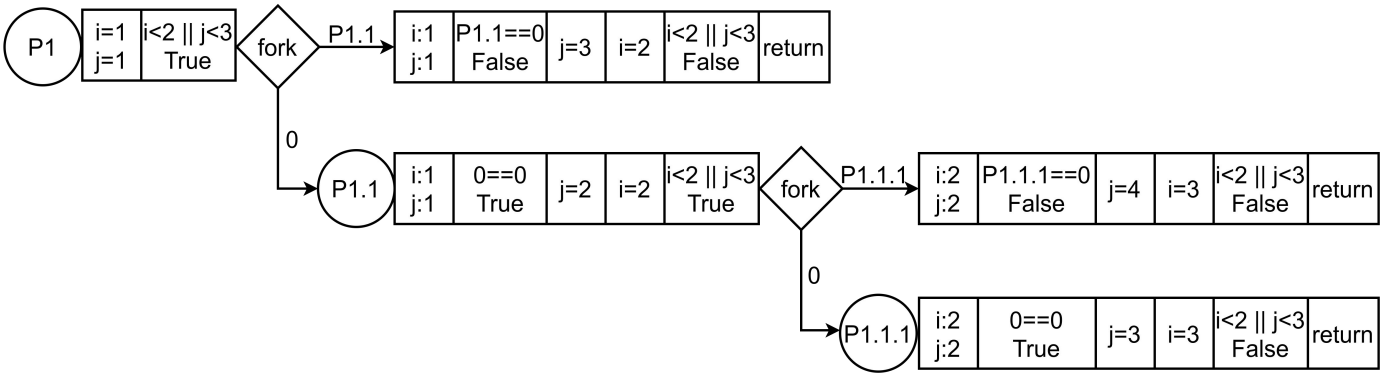
1.2


P1
i=0
j=-1
i=1
fork
P1.1
i:1
j:-1
j=P1.1
j==0
False
i<2
True
i=2
0
P1.1
i:1
j:-1
j=0
j==0
True
i=2
fork
P1.1.1
i:2
j:0
i<2
False
return
0
P1.1.1
i:2
j:0
i<2
False
return
fork
P1.2
i:2
j:P1.1
j=P1.2
j==0
False
i<2
False
return
0
P1.2
i:2
j:P1.1
j=0
j==0
True
i=3
fork
P1.2.1
i:3
j:0
i<2
False
return
0
P1.2.1
i:3
j:0
i<2
False
return

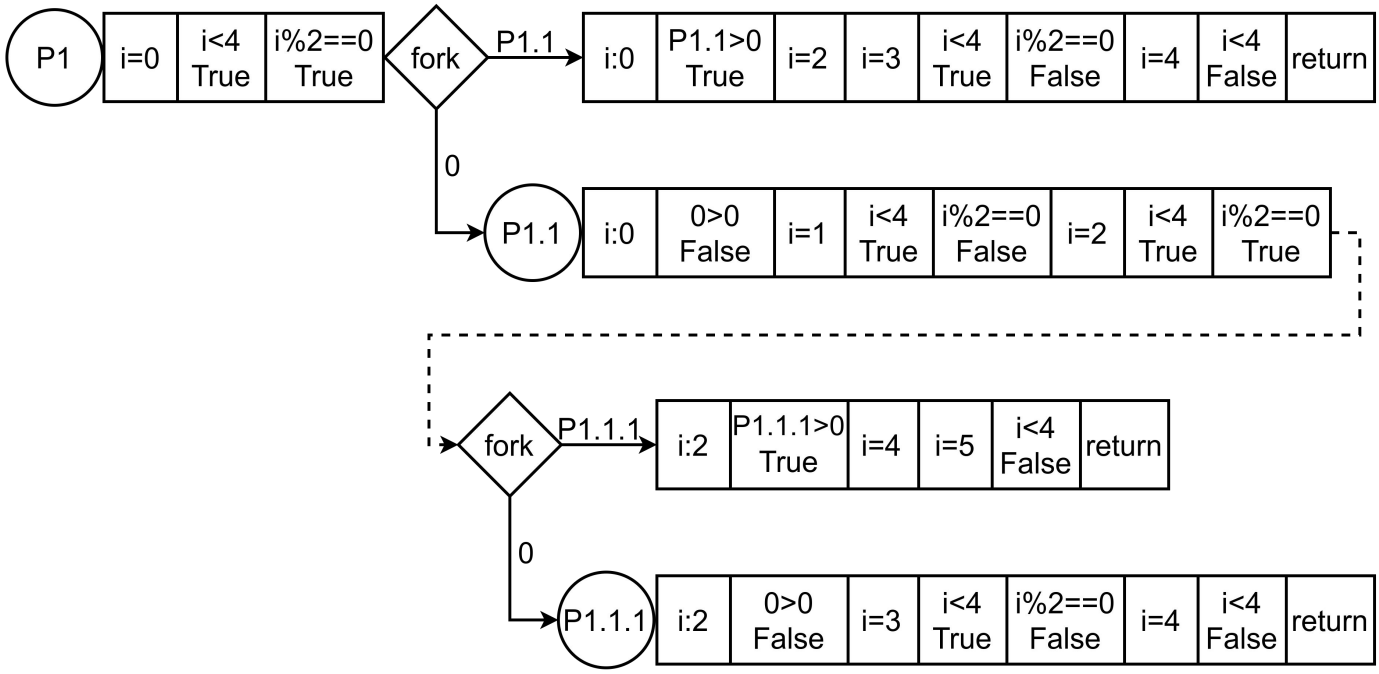
1.3.a


P1
i=1 j=1
i<2 || j<3 True
fork
P1.1
i:1 j:1
P1.1>0 True
j=2
i=2
i<2 || j<3 True
fork
P1.2
i:2 j:2
P1.2>0 True
j=3
i=3
i<2 || j<3 False
return
0
P1.2
i:2 j:2
0>0 False
j=4
i=3
i<2 || j<3 False
return
0
P1.1
i:1 j:1
0>0 False
j=3
i=2
i<2 || j<3 False
return

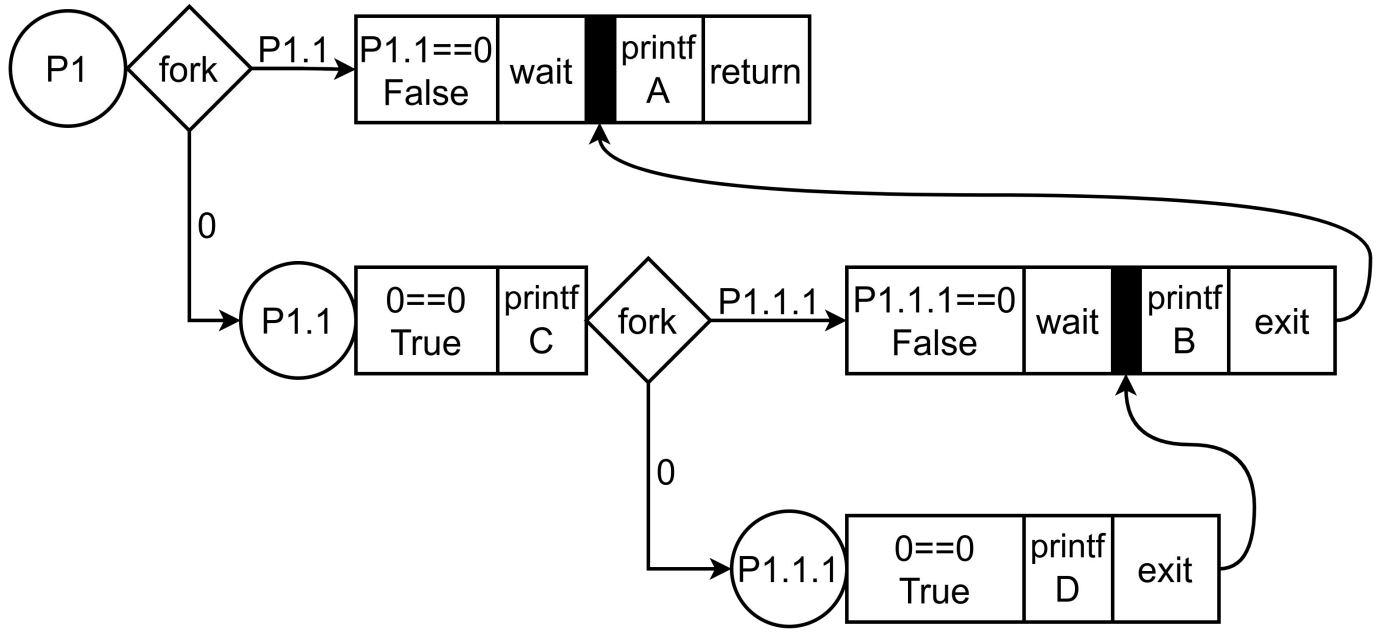
1.3.b


P1
i=1 j=1
i<2 || j<3 True
fork
P1.1
i:1 j:1
P1.1==0 False
j=3
i=2
i<2 || j<3 False
return
0
P1.1
i:1 j:1
0==0 True
j=2
i=2
i<2 || j<3 True
fork
P1.1.1
i:2 j:2
P1.1.1==0 False
j=4
i=3
i<2 || j<3 False
return
0
P1.1.1
i:2 j:2
0==0 True
j=3
i=3
i<2 || j<3 False
return

1.4


P1
i=0
i<4 True
i%2==0 True
fork
P1.1
i:0
P1.1>0 True
i=2
i=3
i<4 True
i%2==0 False
i=4
i<4 False
return
0
P1.1
i:0
0>0 False
i=1
i<4 True
i%2==0 False
i=2
i<4 True
i%2==0 True
fork
P1.1.1
i:2
P1.1.1>0 True
i=4
i=5
i<4 False
return
0
P1.1.1
i:2
0>0 False
i=3
i<4 True
i%2==0 False
i=4
i<4 False
return

1.5


P1
fork
P1.1
P1.1==0 False
wait
printf A
return
0
P1.1
0==0 True
printf C
fork
P1.1.1
P1.1.1==0 False
wait
printf B
exit
0
P1.1.1
0==0 True
printf D
exit

// 1.6

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

main()
{
 pid_t original_parent;

 original_parent=getpid();

 if (fork() == 0) {
   printf("CHILD: before getppid\n");

   // CHILD waits until original PARENT ends
   while(getppid()==original_parent)
     printf("CHILD: original PARENT %d is still alive\n", original_parent);

   printf("CHILD: after getppid: my PARENT is now %d\n", getppid());
   exit(0);
 }

 printf("PARENT: before getchar\n");
 getchar(); // what could happen without "getchar" ?
 printf("PARENT: after getchar\n");
}
```

```
// 1.7.a

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define N 10

main()
{
 int A[N]={0,1,2,3,4,5,6,7,8,9}, i, ret;

 for (i=0; i<N; i++) {
      ret=fork();
      if (ret == 0) {
            if (A[i] % 2 == 0)
                  printf("A[%d]: %d : even\n", i, A[i]);
            else
                  printf("A[%d]: %d : odd\n", i, A[i]);
            exit(0);
      }
 }
```

```
}// 1.7.b

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
 int N, *A, i, ret;

 scanf("%d", &N);
 A=(int *)malloc(N*sizeof(int));

 srandom(getpid());
 for (i=0; i<N; i++)
      A[i] = random();

 for (i=0; i<N; i++) {
      ret=fork();
      if (ret == 0) {
            if (A[i] % 2 == 0)
                  printf("A[%d]: %d : even\n", i, A[i]);
            else
                  printf("A[%d]: %d : odd\n", i, A[i]);
            free(A);
            exit(0);
      }
 }

 free(A);
}
```

```
// 1.7.c

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

main()
{
 int N, *A, i, ret;

 scanf("%d", &N);
 A=(int *)malloc(N*sizeof(int));

 srandom(getpid());
 for (i=0; i<N; i++)
     A[i] = random();

 for (i=0; i<N; i++) {
     ret=fork();
     if (ret == 0) {
          if (A[i] % 2 == 0) {
               printf("A[%d]: %d : even\n", i, A[i]);
               free(A); exit(0);
          }
          else {
               printf("A[%d]: %d : odd\n", i, A[i]);
               free(A); exit(1);
          }
     }
 }

 int status, odds=0;
 for (i=0; i<N; i++) {
      wait(&status);
      ret=WEXITSTATUS(status);
      if (ret == 1) odds++;
 }
 printf("odds: %d\n", odds);

 free(A);
}
```

// 1.8

```
#include <stdio.h>
// block 1 (additional headers files)
#include <stdlib.h>
#include <unistd.h>

void generate_fibonacci(int N, long int *F)
{
 int n;
 F[0]=0; F[1]=1;
 for (n=2; n<N; n++) F[n] = F[n-1] + F[n-2];
}

void show_fibonacci(int N, long int *F)
{
 int n;
 for (n=0; n<N; n++) printf("%ld ", F[n]);
 printf("\n");
}

int main()
{
 int N, i; long int *F;

 scanf("%d", &N);
 // block 2 (code)
 F=(long int*)malloc(N*sizeof(long int));
 generate_fibonacci(N, F);
 show_fibonacci(N, F);

 for (i=0; i<N; i++) {
     // block 3 (code)
     if (fork() == 0) {
          if (F[i] % 2 !=0) printf("%ld\n", F[i]);
          exit(0);
     }
 }

 return(0);
}
```

// 1.9

```
#include <stdio.h>
// block 1 (additional headers files)
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void generate_fibonacci(int N, long int *F)
{
 int n;
 F[0]=0; F[1]=1;
 for (n=2; n<N; n++) F[n] = F[n-1] + F[n-2];
}

void show_fibonacci(int N, long int *F)
{
 int n;
 for (n=0; n<N; n++) printf("%ld ", F[n]);
 printf("\n");
}

int main()
{
 int N, i; long int *F;
 int status, total_odds=0;

 scanf("%d", &N);
 // block 2 (code)
 F=(long int*)malloc(N*sizeof(long int));
 generate_fibonacci(N, F);
 show_fibonacci(N, F);

 for (i=0; i<N; i++) {
     // block 3 (code)
     if (fork() == 0)
          exit(F[i] % 2);
 }

 // block 4 (code)
 while(wait(&status)!=-1)
     total_odds += WEXITSTATUS(status);
 printf("total_odds: %d\n", total_odds);

 return(0);
}
```

// 1.10

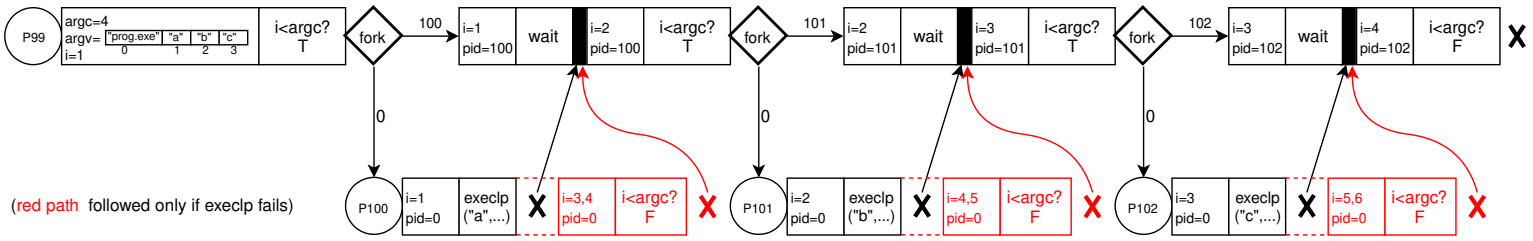
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
// block 1 (additional headers files)
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int i, num_pids_even=0, num_pids_odd=0;
  pid_t pid, pids_even[10], pids_odd[10];
  // block 2 (additional variables)

  // block 3 (code)

  for (i=0; i<10; i++) {
    pid=fork();

    // block 4 (code)

    if (pid == 0) {
          printf("CHILD %d\n", getpid());
          exit(0);
     }
     else if (pid % 2 == 0) {
          pids_even[num_pids_even] = pid;
          num_pids_even ++;
     }
     else {
          pids_odd[num_pids_odd] = pid;
          num_pids_odd ++;
     }
  }

  // block 5 (code)

  printf("PARENT: waiting for even CHILDREN\n");
  for (i=0; i<num_pids_even; i++) {
     pid=waitpid(pids_even[i], NULL, 0);
     printf("PARENT: CHILD %d died\n", pid);
  }

  printf("PARENT: waiting for odd CHILDREN\n");
  for (i=0; i<num_pids_odd; i++) {
     pid=waitpid(pids_odd[i], NULL, 0);
     printf("PARENT: CHILD %d died\n", pid);
  }

  return(0);
}
```

```
// 1.11

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
// block 1 (additional headers files)
#include <stdlib.h>
#include <sys/wait.h>

int main() {
  int i;
  pid_t pids[10];
  // block 2 (additional variables)

  // block 3 (code)

  for (i=0; i<10; i++) {
    pids[i]=fork();

    // block 4 (code)

    if (pids[i] == 0) {
          printf("CHILD %d\n", getpid());
          exit(0);
     }
  }

  // block 5 (code)

  printf("PARENT: waiting for even CHILDREN\n");
  for (i=0; i<10; i++) {
     if (pids[i] % 2 == 0) {
          waitpid(pids[i], NULL, 0);
          printf("PARENT: CHILD %d died\n", pids[i]);
     }
  }

  printf("PARENT: waiting for odd CHILDREN\n");
  for (i=0; i<10; i++) {
     if (pids[i] % 2 != 0) {
          waitpid(pids[i], NULL, 0);
          printf("PARENT: CHILD %d died\n", pids[i]);
     }
  }

  return(0);
}
```

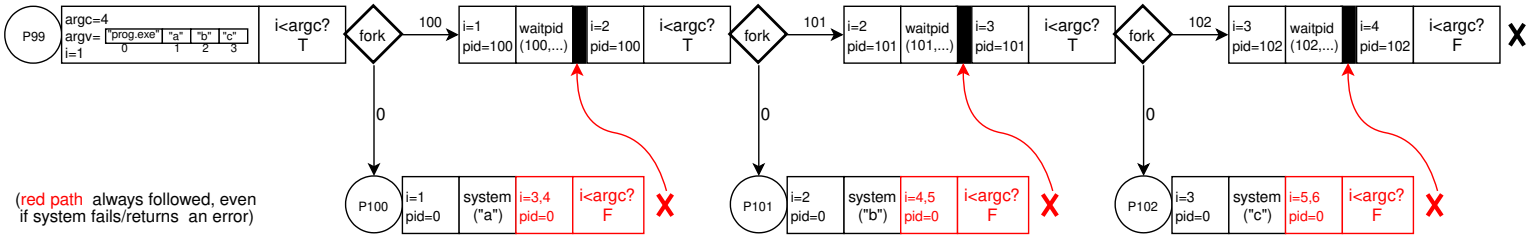
// 1.12

```
#include <stdio.h>

main(int argc, char *argv[])
{
 int i;

 for (i=0; i<=argc-1; i++)
     printf("argv[%d] : %s\n", i, argv[i]);

}
```

// 1.13.a (old notation)



// 1.13.b (old notation)



// 1.14.a

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

main()
{
 pid_t pid;
 char command[256];

 scanf("%s", command);
 pid=fork();

 if (pid==0) {
        printf("CHILD will exec \" %s \"\n", command);
        execlp(command, command, NULL);
        perror("execlp");
        exit(errno);
 }

 printf("PARENT after the child creation\n");
 // ... (other parent tasks)
}
```

// 1.14.b

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

main(int argc, char *argv[])
{
 pid_t pid;
 char *command;

 command=argv[1];
 pid=fork();

 if (pid==0) {
        printf("CHILD will exec \" %s \"\n", command);
        execlp(command, command, NULL);
        perror("execlp");
        exit(errno);
 }

 printf("PARENT after the child creation\n");
 // ... (other parent tasks)
}
```

```
// 1.14.c1
// #include directives equal to those of program 1.13.a

main(int argc, char *argv[])
{
 pid_t pid; int i;
 char *command;

 command=argv[1];
 pid=fork();

 if (pid==0) {
        printf("CHILD will exec \"");
        for (i=1; i<= argc-1; i++) printf(" %s", argv[i]);
        printf(" \"\n");
        switch(argc) {
            case 2: execlp(command, argv[1], NULL);
                    break;
            case 3: execlp(command, argv[1], argv[2], NULL);
                    break;
            case 4: execlp(command, argv[1], argv[2], argv[3], NULL);
                    break;
            // . . .
        }
        perror(execlp);
        exit(errno);
 }

 printf("PARENT after the child creation\n");
 // ... (other parent tasks)
}
```

```
// 1.14.c2
// #include directives equal to those of program 1.13.a

main(int argc, char *argv[])
{
 pid_t pid; int i;
 char *command, **parameters;

 command=argv[1]; parameters=&(argv[1]);
 pid=fork();

 if (pid==0) {
        printf("CHILD will exec \"");
        for (i=1; i<= argc-1; i++) printf(" %s", argv[i]);
        printf(" \"\n");
        execvp(command, parameters);
        perror("execvp");
        exit(errno);
 }

 printf("PARENT after the child creation\n");
 // ... (other parent tasks)
}
```

```
// 1.15

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

main(int argc, char *argv[])
{
 pid_t pid; int i; char *command;

 for(i=1; i<=argc-1; i++) {
        printf("PARENT is going to create a CHILD\n");
        command=argv[i];
        pid=fork();
        if (pid==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
 }

 printf("PARENT after the child(s) creation\n");
 // ... (other parent tasks)
}
```

```
// 1.16

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

main(int argc, char *argv[])
{
 pid_t pid; char *command;

 command=argv[1];
 pid=fork();

 if (pid==0) {
        printf("CHILD will exec %s\n", command);
        execlp(command, command, NULL);
        perror("execlp");
        exit(errno);
 }

 printf("PARENT waiting until CHILD ends\n");
 wait(NULL); // or waitpid(pid, NULL, 0);
 printf("PARENT after CHILD ended\n");
 // ... (other parent tasks)
}
```

```
// 1.17.a
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

main(int argc, char *argv[])
{
 pid_t pid; int i; char *command;

 for(i=1; i<=argc-1; i++) {
        command=argv[i];
        pid=fork();
        if (pid==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
 }

 printf("PARENT waiting until all CHILDREN end\n");
 while(wait(NULL)!=-1);
 printf("PARENT after all CHILDREN ended\n");
 // ... (other parent tasks)
}
```

```
// 1.17.b
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>

main(int argc, char *argv[])
{
 pid_t *pids; int i; char *command;

 pids=(pid_t*)malloc((argc-1)*sizeof(pid_t));

 for(i=1; i<=argc-1; i++) {
        command=argv[i];
        pids[i-1]=fork();
        if (pids[i-1]==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
 }

 printf("PARENT waiting until all CHILDREN end\n");
 for(i=1; i<=argc-1; i++) waitpid(pids[i-1], NULL, 0);
 printf("PARENT after all CHILDREN ended\n");
 // ... (other parent tasks)
 free(pids);
}
```

```
// 1.18.a

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

main(int argc, char *argv[])
{
 pid_t pid; int i; char *command;

 for(i=1; i<=argc-1; i++) {
        command=argv[i];
        pid=fork();
        if (pid==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
 }

 printf("PARENT waiting until last CHILD ends\n");
 waitpid(pid, NULL, 0);
 printf("PARENT after last CHILD ended\n");
 // ... (other parent tasks)
}
```

```
// 1.18.b

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

main(int argc, char *argv[])
{
 pid_t pid; int i; pid_t pid_penultimate; char *command;

 for(i=1; i<=argc-1; i++) {
        command=argv[i];
        pid=fork();
        if (pid==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
        if (i==argc-2)
                pid_penultimate=pid;
 }

 printf("PARENT waiting until penultimate CHILD ends\n");
 waitpid(pid_penultimate, NULL, 0);
 printf("PARENT after penultimate CHILD ended\n");
 // ... (other parent tasks)
}
```

```
// 1.19

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

main(int argc, char *argv[])
{
 pid_t pid; int i;
 char *command;

 for(i=1; i<=argc-1; i++) {
        command=argv[i];
        pid=fork();
        if (pid==0) {
                printf("CHILD will exec %s\n", command);
                execlp(command, command, NULL);
                perror("execlp");
                exit(errno);
        }
        wait(NULL); // or waitpid(pid, NULL, 0);
 }

 printf("PARENT after all CHILDREN ended\n");
 // ... (other parent tasks)
}
```

```
// 1.20
// the behavior of "mysystem" tries to be similar,
// as much as possible, to the behavior of "system",
// accordingly with the man page of this primitive

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

int mysystem (char *command)
{
 pid_t pid; int state;

 pid=fork();
 if (pid<0) return(errno);
 if (pid == 0) {
     execl("/bin/sh", "sh", "-c", command, NULL);
     exit(127);
 }
 waitpid(pid, &state, 0);
 return(WEXITSTATUS(state));
}

main()
{
 mysystem("ls -la");
 printf("after mysystem\n");
}
```

// 1.21

```
#include <stdio.h>
// block 1 (additional headers files)
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
 char Operations[8][2]={"+",".","+",".","-",".","-","."};
 int A[8]={1,2,3,4,5,6,7,8}, B[8]={8,7,6,5,4,3,2,1}, i;
 // block 2 (more variables)
 int status, result;

 for (i=0; i<8; i++) {

   // block 3 (code)

   if (fork()==0) {
     char num1[10],num2[10];

     sprintf(num1,"%d",A[i]); sprintf(num2,"%d",B[i]);

     execl("./mycalc.exe","mycalc.exe",Operations[i],num1,num2, NULL);

    exit(0);
   }
 }

 // block 4 (code)

 result=0;
 while(wait(&status)!=-1)
   result += WEXITSTATUS(status);

 printf("result: %d\n", result);
 //84 = (1+8)+(2*7)+(3+6)+(4*5)+(5-4)+(6*3)+(7-2)+(8*1)
}
```