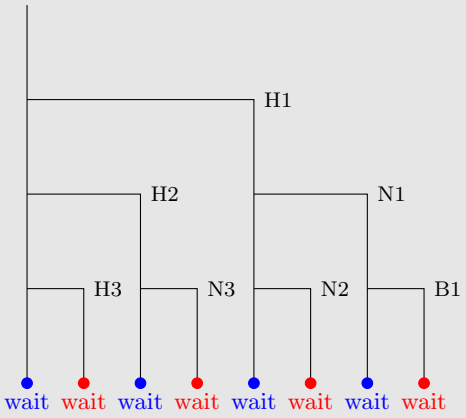
1. Dado el siguiente código, realiza el diagrama que refleja los procesos creados. Explica a través del diagrama cómo finalizan los procesos, si quedan procesos zombies o huérfanos y cuáles son si los hubiese.
   Nota: Sólo por simplificar, no se está controlando que la llamada a la función *fork()* pueda devolver error. Como buenas prácticas de programación se debería incluir la comprobación de posible error.

```
main(){
    fork();
    fork();
    fork();
    wait();
}
```

Solución:



A modo de ilustración, este primer ejemplo se explicará con todo detalle.

- Con el primer fork() se lanza un proceso hijo (**H1**).
- En el segundo fork(), tanto el proceso padre como el proceso hijo lanzan cada uno un proceso y por tanto hay un proceso padre (**P**), dos procesos hijos (**H1** y **H2**) y un proceso nieto (**N1**).
- En el tercer fork() el proceso padre, los dos procesos hijos y el proceso nieto lanzan un nuevo proceso cada uno de ellos. Así quedan un proceso padre (**P**), tres procesos hijos (**H1**, **H2** y **H3**), tres procesos nietos (**N1**, **N2** y **N3**, dos descendientes de **H1** y uno descendiente de **H2**) y un proceso biznieto (**B1**, descendiente del abuelo **H1**).
- Cada uno de los ocho procesos hace una llamada a la función wait(). Como se puede ver en la figura:
  - El proceso biznieto (**B1**), dos nietos (**N2** y **N3**) y el proceso hijo (**H3**) no tienen descendientes. De esta forma, la llamada a la función wait() no produce error ni se bloquea y los cuatro procesos pueden finalizar quedando en estado zombie hasta que su proceso padre pueda recogerlo y se libere definitivamente. *Metafóricamente, imagina un niño esperando en la puerta del colegio, una vez finalizada la jornada escolar (liberación de los recursos y memoria), hasta que su padre pasa a recogerlo (finalización del proceso).*
  - De igual forma el proceso hijo **H2** y el proceso nieto **N1** hacen una llamada a la función wait() y como sólo tienen un descendiente y es uno de los anteriores (**N2** o **N3**, según el orden de ejecución, para el proceso **H2** y **B1** para el proceso **N1**), recogerán a sus

procesos hijos (*estos son los padres que recogen a los niños que están esperando en el colegio*). A su vez los procesos **H2** y **N1** enviarán la notificación a su padre de terminación. **H2** y **N1** pueden quedar en estado zombie o huérfano dependiendo de la dinámica de ejecución de sus procesos padre.

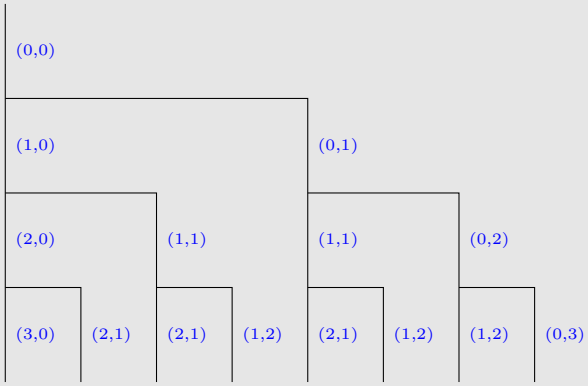
- El proceso hijo **(H1)** tiene dos descendientes (**N1** y (**N2** o **N3**, según el orden de ejecución)) y una única llamada a la función wait(), solo podrá finalizar completamente uno de los procesos. El otro proceso quedará huérfano, siendo el proceso init el que finalmente termine la ejecución del proceso huérfano.
- El proceso padre tiene tres descendientes y un sólo wait(). Como sus descendientes terminan correctamente eso implica que al menos dejará dos de sus descendientes huérfanos que serán adoptados por el proceso init.

La velocidad relativa de los procesos es la que va a determinar lo que realmente ocurra. A priori, no podemos decir con exactitud qué proceso va a terminar primero y cuál el último.

2. Realiza un diagrama con los procesos creados a partir de la ejecución del siguiente código. Identifica cada proceso con una letra e indica lo que imprimirán por la terminal cada uno de los procesos. Indica también si quedan procesos zombies o huérfanos y si se puede determinar cuáles.

```
main(){
    static int a[2]={0,0};
    static int c = 0;

    if(c == 3) exit(0);
    ++c;

    if (fork()) ++a[0];
    else ++a[1];
    main();

    printf ("(%d,%d)",a[0],a[1]);

    if(c != 3) wait();
}
```

Solución:



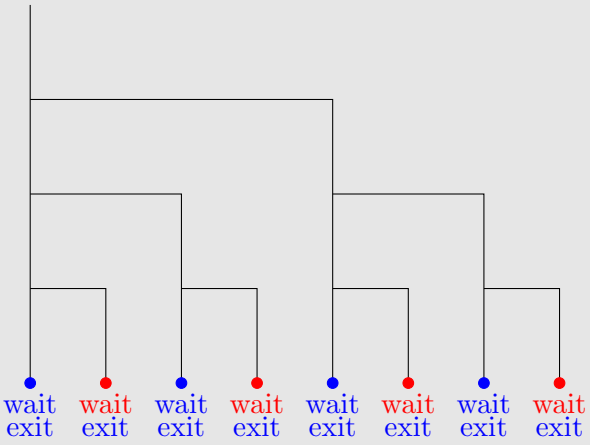
Todos los procesos generados, incluido el padre ejecutan la sentencia if(c == 3) exit(0) y por tanto ninguno ejecuta ningún wait(). Esto hará que todos todos los procesos salvo el padre puedan convertirse en zombies o huérfanos dependiendo de la dinámica de los procesos.

No se imprimirá nada porque nunca llegará a la sentencia printf(). Las dos últimas sentencias nunca se ejecutan, sin embargo en el dibujo, para una mayor comprensión, se ponen los valores del vector a.

3. Dibuja el árbol de procesos y determina qué procesos quedan zombies o huérfanos para el siguiente código:

```
main(){
    static int i=0;

    if(i != 3){
        ++i;
        fork();
        main();
    } else wait();
    exit(0);
}
```

Solución:

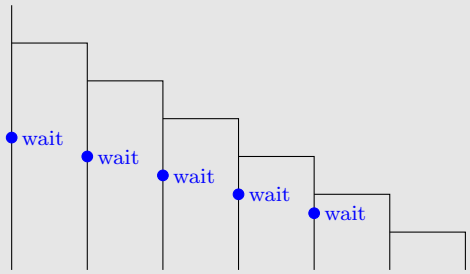


Todos los procesos generados realizan un wait(). Los procesos que no tienen hijos el sistema operativo detecta que no tienen hijos y por tanto el wait ni se bloquea y retorna con -1 y el hijo termina correctamente. Todos los procesos que sólo tienen un hijo finalizan correctamente y sus hijos no dejan a ningún proceso ni huérfano ni zombie. Los procesos que tienen más de un hijo dejan a todos sus procesos hijos menos a uno en estado de huérfano o de zombie dependiendo de la velocidad relativa de los procesos.

4. Dibuja un esquema en forma de árbol de los procesos que se crean según el código que se indica más abajo. Explica también si los procesos terminan correctamente o no lo hacen y por qué motivos.

```
main(){
    for (i=0; i< 6; ++i)
        if(fork()) break;
    if(i<5) wait();
    exit();
}
```

Solución:



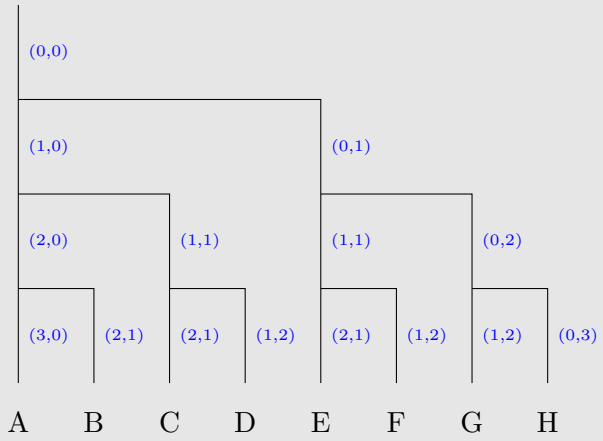
Todos los procesos menos los dos últimos realizan un wait antes de terminar y por tanto sus hijos terminarán correctamente aunque puedan estar temporalmente en el estado de zombies. Todos los procesos terminan correctamente menos el último que se queda huérfano o zombie dependiendo de la velocidad relativa de los procesos.

5. Se realiza el código adjunto. Realiza un diagrama con los procesos creados. Identifica cada proceso con una letra e indica lo que imprimirán por la terminal cada uno de los procesos.

```
main(){
    int a[2];

    a[0] = a [1] = 0;
    for ( i = 0; i < 3; ++i){
        if (fork()) ++a[0];
        else ++a[1];
    }
    printf ("(%d,%d)",a[0],a[1]);
}
```

Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc.

Solución:



Todos los procesos se quedan zombies menos el padre. Pueden haber estado incluso en el estado de huérfano si el padre de cada hijo termina antes que el hijo.

Lo que imprime cada proceso es lo siguiente:
A: 3,0; **B**: 2,1; **C**: 2,1; **D**: 1,2; **E**: 2,1; **F**: 1,2; **G**: 1,2; **H**: 0,3.

6. Se realiza el código adjunto. Realiza un diagrama con los procesos creados.

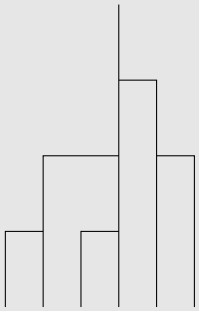
```
int main(){
     if( !fork()){
         fork();
         wait();
     } else {
         fork();
         fork();
         wait();
     }
     exit(EXIT_SUCCESS);
 }
```

Se pide:

A. Dibuja la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:



Todos los procesos realizan un wait. Los procesos que no tienen hijos su wait devuelve -1 y finalizan. Los procesos que sólo tienen un hijo realizan el wait de su hijo y por tanto finalizarán correctamente. El padre es el único proceso que tiene más de un hijo y como sólo realiza un wait recogerá correctamente al hijo que finalice primero pero los otros dos procesos pueden quedar huérfanos o zombies dependiendo de la velocidad relativa de los procesos.

7. Se realiza el código adjunto. Realiza un diagrama con los procesos creados. Identifica cada proceso con una letra e indica lo que imprimirán por la terminal cada uno de los procesos.

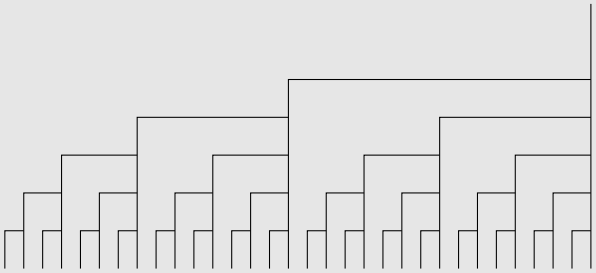
```
int main(){
      int i, pid;

      for ( i = 0; i < 3; i++){
          pid=fork();
          if(!i%2) fork();
      }

      for( i = 0; i < 5 ; ++i) wait();
      exit(EXIT_SUCCESS);
 }
```

Se pide:

A. Dibujar la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:



Todos los procesos realizan cinco waits. Como ningún proceso tiene más de 5 hijos pueden sobrarles waits pero como el sistema operativo sabe que no tienen hijos los wait no se bloquean y los porcesos finalizan correctamente. Si cualquier hijo termina antes de que el padre realice el wait estará temporalmente zombie hasta que su padre realice un wait.

8. Se realiza el código adjunto. Realiza un diagrama con los procesos creados. Identifica cada proceso con una letra e indica lo que imprimirán por la terminal cada uno de los procesos.

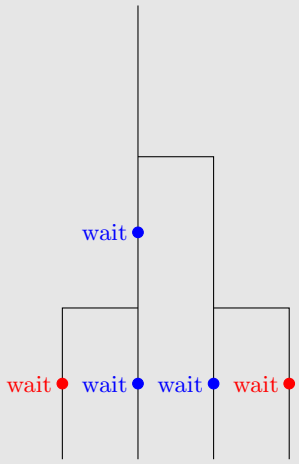
```
main(){
      if(fork()){
          wait();
          if(fork()) wait();
      } else {
          fork();
          wait();
      }
      exit(0);
  }
```

Se pide:

A. Dibuja la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:



Dos procesos wait de más pero como el sistema operativo sabe que no tienen hijos, retorna -1 y los procesos terminan. Los otros dos procesos hacen tantos waits como hijos tienen y por tanto ningún proceso queda huérfano o zombie.

9. Se realiza el código adjunto.

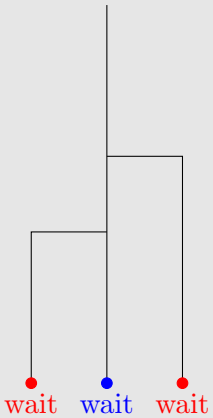
```
main(){
    int i=0;
    pid_t pid;

    while((pid = fork()) != 0 && i < 1){
        if(pid) fork();
        else wait();
        ++i;
    }
    wait();
}
```

Se pide:

A. Dibuja la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:

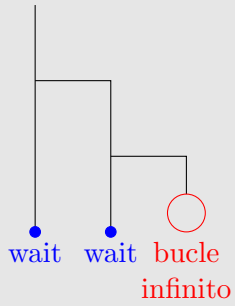


Los dos hijos hacen un wait y como no tienen hijos devuelve -1 y el proceso puede continuar. Sin embargo el padre sólo hace un wait para dos hijos. El primero que termine finalizará correctamente y el segundo podrá estar huérfano to zombie dependiendo de la velocidad relativa de los procesos.

10. Imagina que se ejecuta el siguiente código. Realiza un diagrama con los procesos creados. Explica a través del diagrama el estado de los distintos procesos.

```
main() {
     if (fork()) {
          wait();
          while(fork()) fork();
     }
     else {
          if (fork())
               wait();
          else
               while(TRUE);
     }
}
```

Solución:



El proceso padre se bloquea en el wait esperando a su hijo. El hijo lanza a un nieto y se bloquea en el wait a la espera de que finalice el nieto. El nieto nunca finaliza porque tiene un bucle infinito. Por tanto el padre y el hijo están bloqueados a la espera de un nieto que está en un bucle infinito. Como el padre no puede continuar nunca se realizará resto del código de lanzamiento de hijos que además es un fork-bomb con infinitos procesos.

11. Se realiza el código adjunto.

```
main()
{
    int i=0;
    pid_t pid;

    while((pid = fork()) != 0 && i < 1)
    {
        if(pid) fork();
        else wait();
        ++i;
```

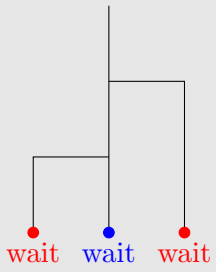
```
        }
        wait();
    }
```

Se pide:

A. Dibuja la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:



No queda ningún proceso zombie pero se producen dos waits que devuelven error puesto que en el momento de realizarlos el proceso no tiene ningún hijo. Estos son los marcados en rojo.

12. Se realiza el código adjunto.

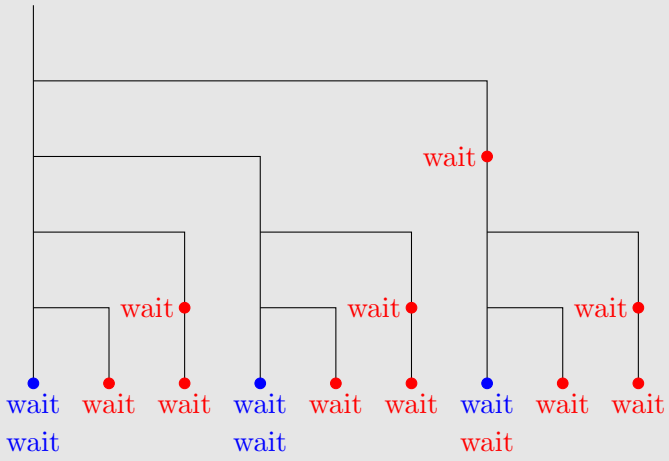
```
main()
{
    int i=0;
    pid_t pid;

    while(i < 2)
    {
        pid = fork();
        if(pid) fork();
        else wait();
        ++i;
    }
    if(pid) wait();
    wait();
}
```

Se pide:

A. Dibuja la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. De existir algún problema diseño (procesos huérfanos, procesos zombies, etc ...) indica y explica este problema de diseño y qué proceso o procesos se pueden ver afectados.

Solución:



Todos los wait que están en rojo devolverán -1 y continuarán dado que los procesos no tienen ningún hijo al que esperar. Por otro el padre principal sólo hace dos waits y tiene cuatro hijos con lo que forzosamente dos quedarán huérfanos o zombies dependiendo de la velocidad relativa de los procesos, así mismo hay un proceso que realiza tres waits y sólo tiene dos hijos por lo que el último wait devolverá -1 y continuará.

13. Se realiza el código adjunto. Realiza un diagrama con los procesos creados. Identifica cada proceso con una letra e indica lo que imprimirán por la terminal cada uno de los procesos.

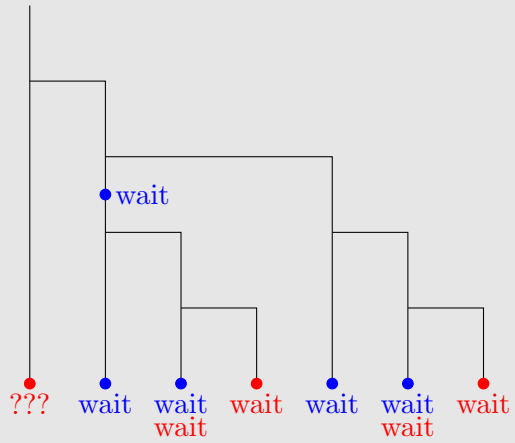
```
main()
{
    int i=0;

    while(!fork())
    {
        if(fork()) wait();
        wait();
        if(i == 1) break;
        else i++;
    }
}
```

Se pide:

A. Dibujar la jerarquía de procesos que resulta de la ejecución de dicho código. Identifica cada proceso con una letra o numeración distinta. ¿Cuántos procesos, contando el proceso padre, se habrán generado en la ejecución del código?

B. Indica qué problema de diseño tiene el código presentado. Procesos huérfanos, procesos zombies, etc ...

Solución:



El padre de todos deja a su hijo huérfano o zombie dependiendo de las velocidades relativas de los procesos.

El resto de los procesos realizan un wait a todos sus hijos y realizan waits de más que retornan con -1 y continúan. Estos son los waits en rojo de la figura anterior.

14. Se dispone del siguiente código:

```
main(){
    char d = 'A';

    for( i = 0 ; i < 2 ; ++i){
        ++d;
        if (fork()){
            ++d;
            wait();
            if(fork()){
                ++d;
                wait();
            } else printf("Soy %c\n",d);
        } else printf("Soy %c\n",d);
    }
}
```

A. ¿Qué salida se produce en la terminal? Soy B
Soy C
Soy C
Soy D
Soy D
Soy E
Soy E
Soy D

B. ¿Quedan procesos huerfanos o zombies? Justifica la respuesta Justo tras cada fork el padre realiza un wait luego todos los padres esperan a que acaben sus hijos antes de imprimir su salida. Por ese motivo las finalizaciones son ordenadas y no sobra ni falta ningún wait.

15. Dado el siguiente fragmento de código y suponiendo que no hay errores en la ejecución de la llamada a la función fork(). 1

```
pidA = fork(); printf("A");
wait(pidA);
pidB = fork(); printf("B");
pidC = fork(); printf("C");
wait(pidC);
wait(pidB);
exit(EXIT_SUCCESS);
```

A. Dibuja un esquema o diagrama indicando qué procesos se crean con el fragmento de código anterior.

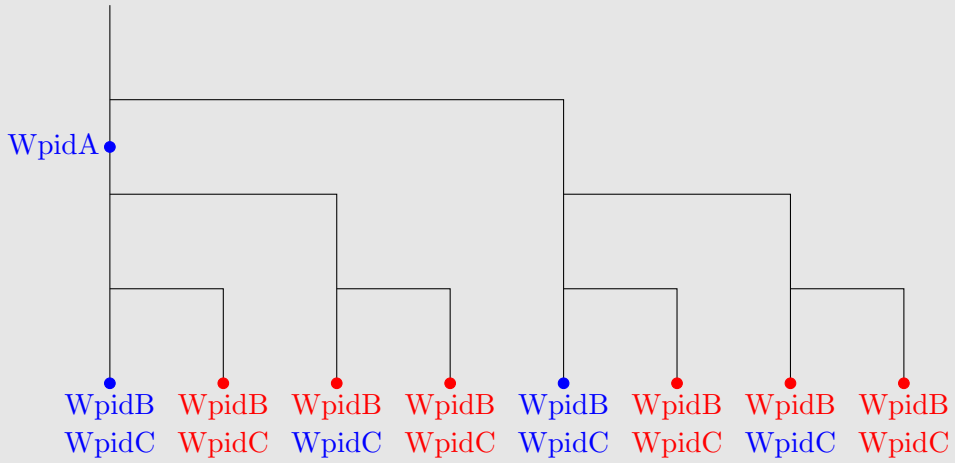
B. Relativo a las letras A, B, C y D, ¿cuántas de cada una de ellas aparecerán en pantalla?

C. Dichas letras, ¿aparecerán en algún orden determinado?. Si la respuesta fuera positiva, escribe el orden. En cualquier caso razona la respuesta.

D. ¿Se producen procesos huerfanos o zombies? Razona la respuesta.

Solución:

A.



B. Imprimirá 2 A, 4 B y 8 C.

C. Las 2 A siempre aparecerán primeras después siempre aparecerá una B al menos. A partir de ese momento por cada B que se imprima permitirá la impresión de dos C aunque la combinación de B y C puede ser cualquiera mientras se cumpla la regla anterior, eso fuerza a que las dos últimas sean C. Ejemplos de combinaciones posibles: AABBCCBCBCCCCC, AABCBCBCBCCCCC o AABBBCCCCCCBCC.

D. En el dibujo pueden verse en rojo los Waits que no tienen efecto y por tanto retornan -1 y el proceso continúa. Sobran muchos y sin embargo están todos los waits necesarios para que ningún proceso se quede huérfano ni zombie.