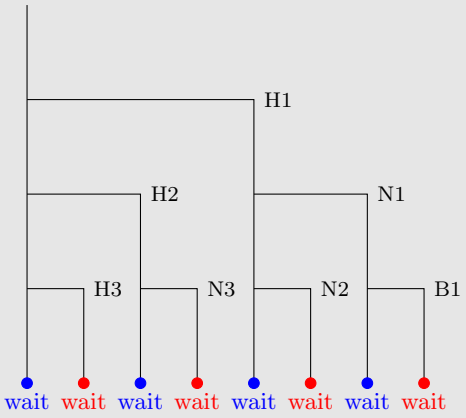
1. Dado el siguiente código, realiza el diagrama que refleja los procesos creados. Explica a través del diagrama cómo finalizan los procesos, si quedan procesos zombies o huérfanos y cuáles son si los hubiese.
   Nota: Sólo por simplificar, no se está controlando que la llamada a la función *fork()* 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() el proceso **P** lanza un proceso hijo (**H1**).
- En el segundo fork(), tanto el proceso padre como el proceso hijo lanzan cada uno un proceso (**H2** y **N1**). Por tanto, en este momento hay un proceso padre (**P**), dos procesos hijos (**H1** y **H2**) y un proceso nieto (**N1**).
- En el tercer fork(), todos los procesos creados hasta el momento lanzan un nuevo proceso cada uno de ellos (**H3**, **N3**, **N2** y **B1**). Por tanto, tenemos un proceso padre (**P**), tres hijos (**H1**, **H2** y **H3**), tres 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:
  - Los procesos **B1**, **N2**, **N3** y **H3** no tienen descendientes. Por tanto, la llamada a la función wait() no produce error ni les bloquea, sino que devuelve -1 y los cuatro procesos pueden finalizar sin problemas, quedando en estado zombie (sin ocupar ya memoria en el sistema, salvo un campo donde se guarda su código de finalización) hasta que sus procesos padres puedan recoger ese código de finalización y se libere también ese campo de memoria. *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).*
  - Por otra parte, los procesos **H2** y **N1** hacen una llamada a la función wait() y como sólo tienen un hijo, recogerán el código de finalización de sus hijos correctamente (*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 de terminación a sus padres. **H2** y **N1** pueden quedar en estado zombie o huérfano dependiendo de la dinámica de ejecución de sus procesos padres, que no ejecutan el número adecuado de waits (1/hijo).

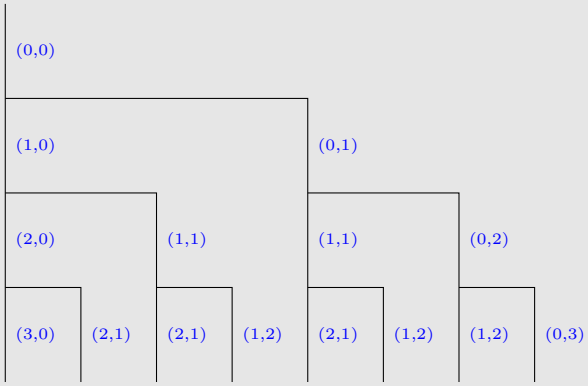
- El proceso **(H1)** tiene dos descendientes (**N1** y **N2**) y una única llamada a la función wait(), por lo que solo podrá finalizar completamente uno de los procesos. El otro proceso quedará huérfano (siendo el proceso init el que lo adopte y finalmente termine la ejecución del proceso huérfano) o zombie, dependiendo de la velocidad relativa entre el hijo y el padre.
- El proceso padre P tiene tres descendientes y un sólo wait(). Como sus descendientes terminan correctamente, al menos dos de ellos quedarán huérfanos y serán adoptados por el proceso init, o zombies, dependiendo de la velocidad relativa entre dichos procesos y su padre.

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 último.

2. Dibuja 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:



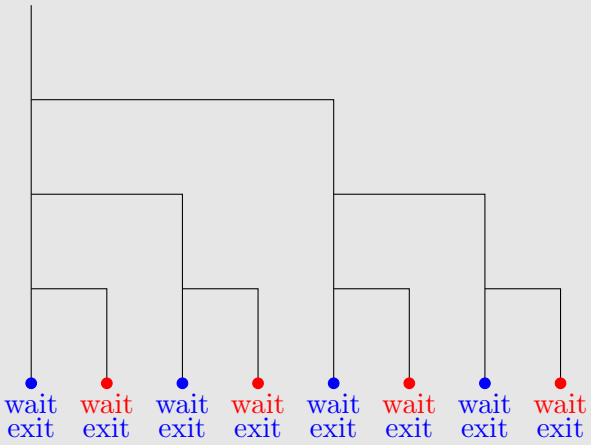
Todos los procesos generados, incluido el padre, ejecutan la sentencia if(c == 3) exit(0) y por tanto ninguno ejecuta ningún wait(). Esto hará que todos todos los procesos salvo el padre puedan convertirse en zombies o huérfanos dependiendo de la dinámica de los procesos.

No se imprimirá nada porque nunca llegará a la sentencia printf(). Las dos últimas sentencias nunca se ejecutan. En el dibujo se han indicado los valores del vector a únicamente para una mayor comprensión.

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:

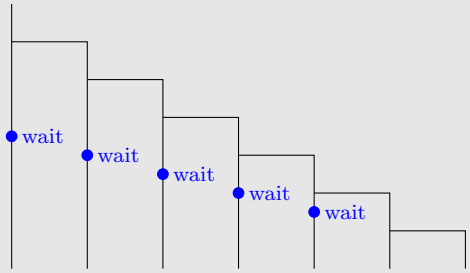


Todos los procesos generados realizan un wait(). Sobre los procesos que no tienen hijos, como el sistema operativo lo sabe, los waits (en rojo) no se bloquean, sino que retornan -1 y esos procesos continuan su ejecución y terminan correctamente. Todos los procesos que sólo tienen un hijo finalizan correctamente, pues hacen 1 wait, y sus hijos no quedan ni huérfanos ni zombies. 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 de debajo. Explica si los procesos terminan correctamente o no y por qué.

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

Solución:



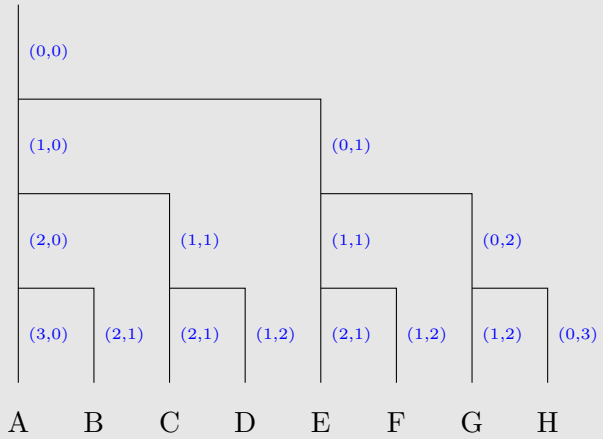
Todos los procesos menos los dos últimos realizan un wait antes de terminar. Por tanto, sus hijos terminarán correctamente, aunque puedan estar temporalmente en el estado de zombies. Todos los procesos terminan correctamente menos el último, porque su padre no recoge su código de finalización. Por tanto, este último proceso quedará huérfano o zombie dependiendo de la velocidad relativa entre él y su padre.

5. Se ejecuta el código adjunto. Dibuja 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 o huérfanos menos el padre. Pueden haber estado huérfanos si el padre de cada hijo termina antes que el hijo. Estarán zombies si el hijo termina antes y envía su código de finalización al padre, que no lo recoge.

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 ejecuta el código adjunto. Dibuja un diagrama con los procesos creados.

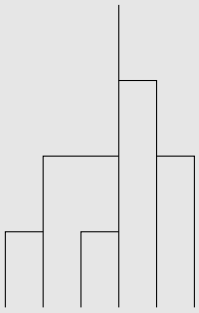
```
int main(){
     if( !fork()){
         fork();
         wait();
     } else {
         fork();
         fork();
         wait();
     }
     exit(EXIT_SUCCESS);
 }
```

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

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

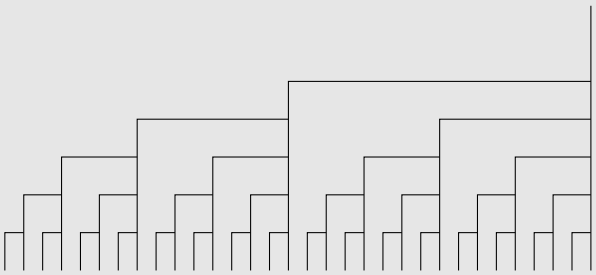
```
int main(){
      int i, pid;

      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. 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 5 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 bloquean a los procesos, y éstos finalizan correctamente. Si cualquier hijo termina antes de que su padre ejecute el wait, estará temporalmente zombie hasta que su padre ejecute el wait.

8. Se ejecuta el código adjunto. Dibuja 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.

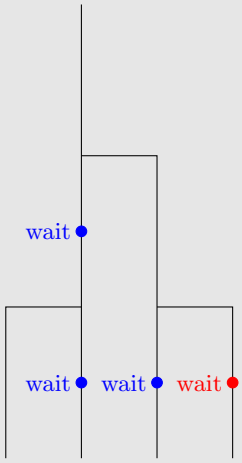
```
main(){
      if(fork()){
          wait();
          if(fork()) wait();
      } else {
          fork();
          wait();
      }
      exit(0);
  }
```

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:



El wait en rojo sobra, pues el proceso que lo ejecuta no tiene hijos. Como el sistema operativo lo sabe, la llamada a wait retorna -1 y el proceso termina sin problemas. El proceso padre de todos y su primer hijo hacen tantos waits como hijos tienen, y los hacen en el momento oportuno (tras haber creado a sus hijos), por lo que ningún proceso queda huérfano o zombie.

9. Se ejecuta el código adjunto.

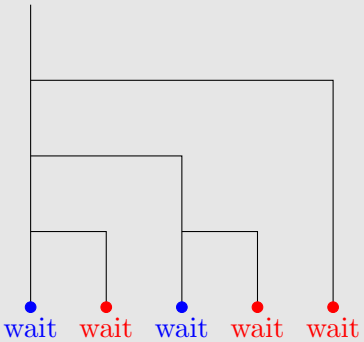
```
main(){
    int i=0;
    pid_t pid;

    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:

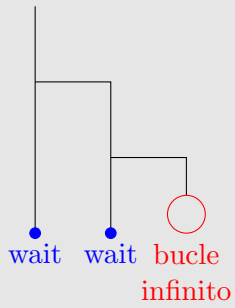


Todos los procesos hacen un wait. Algunos no tienen ningún hijo y el wait no se bloquea y devuelve un error. El padre principal tiene tres hijos y sólo hace un wait con lo que dos de sus hijos quedarán huérfanos o zombies dependiendo de la velocidad relativa de los procesos.

10. Imagina que se ejecuta el siguiente código. Dibuja 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 ejecuta el código adjunto.

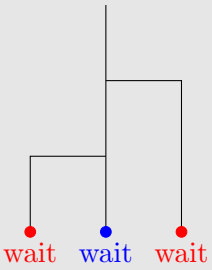
```
main() {
     int i=0;
     pid_t pid;

     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 ejecuta el código adjunto.

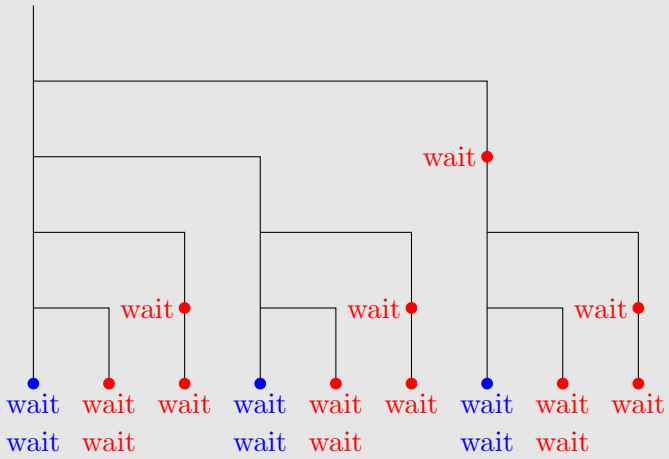
```
main()
{
    int i=0;
    pid_t pid;

    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. Muchos waits fallan puesto que cuando se realizan el proceso carece de hijos. Esos waits devuelven -1 y los procesos continúan.

13. Se ejecuta el código adjunto. Dibuja 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.

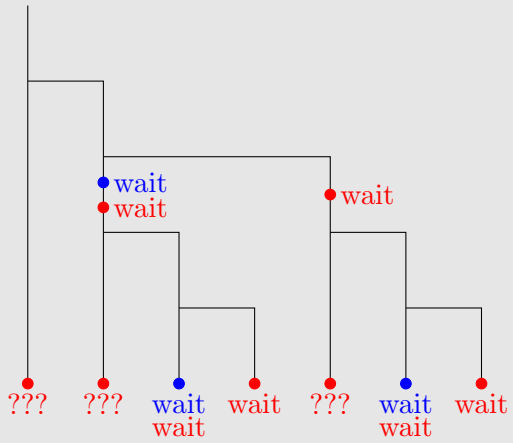
```
main()
{
    int i=0;

    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 esos dos procesos, porque no espera a recoger el código de finalización del hijo tras crearlo. Así mismo, el segundo nieto también quedará huérfano o zombie, porque su padre no espera tras crearle, y el penúltimo proceso de la derecha también, por la misma razón.

Los waits en rojo devuelven -1 porque los procesos que los ejecutan no tienen hijos en ese momento; los procesos que los ejecutan continúan. Los waits en azul esperan a un hijo que han creado.

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 D
Soy C
Soy D
Soy E
Soy E
Soy F

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.

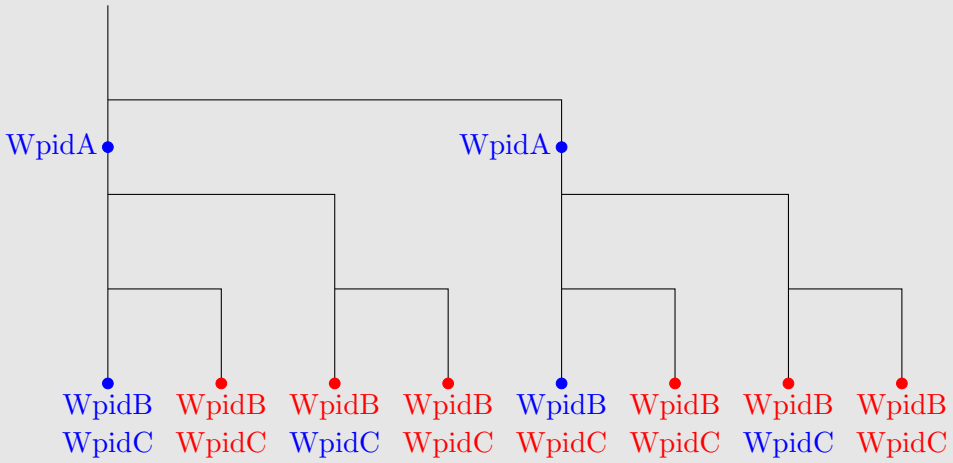
B. Relativo a las letras A, B, C y D, ¿cuántas de cada una de ellas aparecerán en pantalla?

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.