Diseño de Sistemas Operativos. Junio de 2013
Ejercicio de procesos y memoria

Se pretende analizar la implementación de los servicios UNIX vinculados con la terminación de procesos (`_exit(status)` y `pidhijo=wait(&status)`). A continuación, se muestra una versión simplificada, pero incorrecta, de estos servicios para un sistema monoprocesador con un núcleo no expulsivo. Téngase en cuenta que en el BCP estarán definidos todos los campos requeridos para esta funcionalidad.

```
int wait(void *pstatus){
  BCP *phijo_zom; int pidh;
  if (pactual->nhijos==0) return -1;
  if (pactual->nzombis==0) {
    pactual->dirstat = pstatus;
    bloquear(&(pactual->cola_espera_hijo));
    phijo_zom = pactual->hijo_zom;
  }
  else {
    phijo_zom = obtener_hijo_zombi(pactual);
    *(pstatus) = phijo_zom->valexit;
  }
  pactual->nhijos--; pactual->nzombis--;
  pidh = phijo_zom->pid;
  liberar_pila_nucleo(phijo_zom->pila_nuc);
  liberar_BCP(phijo_zom);
  return pidh;
}
```

```
void _exit(int status) {
  BCP *ppadre = pactual->padre;
  liberar_recursos(pactual); //ficheros,mapa,…
  . . . . . . . . . . . . .
  pactual->valexit=(status&0xff)<<8;
  pactual->estado=ZOMBI;
  ppadre->nzombis++;
  if (!vacia(&(ppadre->cola_espera_hijo))) {
    *(ppadre->dirstat) = pactual->valexit;
    ppadre->hizo_zom = pactual;
    desbloquear(&(ppadre->cola_espera_hijo));
  }
  // típico (y correcto) cambio proceso por fin
  . . . . . . . . . . . . . // ops. típicas
  pactual = planificador();
  cambio_contexto(NULL, &(pactual->ctx));
}
```

a) Explique por qué motivo es incorrecta y modifíquela para corregirla.

b) Utilizando la versión corregida, especifique la traza de ejecución del siguiente ejemplo, identificando las activaciones del sistema operativo, los cambios de contexto, distinguiendo entre voluntarios e involuntarios, así como qué tipos de fallos de página se producen suponiendo una situación donde existen marcos de página libres.

- **P** (prioridad alta): invoca una llamada `wait` especificando como parámetro la dirección de una variable global con valor inicial almacenada en una página a la que no se había accedido previamente. Después, ejecuta en modo usuario y termina de forma voluntaria (`_exit`).
- **Q** (prioridad media e hijo único de **P**): Proceso recién creado que ejecuta en modo usuario y termina (`_exit`).
- **R** (prioridad baja): Proceso recién creado que ejecuta todo el tiempo en modo usuario.

c) La liberación de los recursos de un proceso durante la finalización del mismo es una operación delicada puesto que el proceso tiene que desmantelarse a sí mismo pero manteniendo los suficientes recursos para proseguir su ejecución hasta que le cede el procesador a otro proceso. Uno de los recursos más críticos es la pila de núcleo del proceso, pues es un recurso imprescindible para su ejecución en modo núcleo. La solución habitual para evitar problemas en la liberación de ese recurso es que sea otro proceso el que la lleve a cabo una vez terminado el primero. En un sistema UNIX, como se aprecia en el primer apartado, lo habitual es que sea el proceso padre el que realice dicha operación dentro del servicio `wait`. En otros sistemas, en los que un proceso desaparece del sistema en cuanto completa su ejecución, la solución más frecuente es que sea el proceso al que le cede el control el proceso completado el que realice la liberación de la pila de núcleo del primero.

Supóngase que en un determinado sistema monoprocesador con un esquema no expulsivo todos los procesos ejecutan el siguiente fragmento de código (en tres versiones; sólo una de ellas correcta) a la hora de realizar cualquier cambio de contexto, ya sea porque el proceso se ha bloqueado, ha sido expulsado o ha completado su ejecución.

```
// 1ª versión
// variables globales
BCP *actual, *prev;
   . . . . .
   prev=actual;
   actual=planificador();
   if (prev->estado==FIN)
      ccontexto(NULL,actual);
   else
      ccontexto(prev,actual);
   if (prev->estado==FIN)
      liberar_pila(prev);
   . . . . .
```

```
// 2ª versión
// variable global
BCP *actual;
   . . . . .
   BCP *prev; // var. local
   prev=actual;
   actual=planificador();
   if (prev->estado==FIN)
      ccontexto(NULL,actual);
   else
      ccontexto(prev,actual);
   if (prev->estado==FIN)
      liberar_pila(prev);
   . . . . .
```

```
// 3ª versión
// variable global
BCP *actual;
   . . . . .
   BCP *prev, *pos;//locales
   prev=actual;
   pos=actual=planificador();
   if (prev->estado==FIN)
      ccontexto(NULL,actual);
   else
      ccontexto(prev,actual);
   if (pos->estado==FIN)
      liberar_pila(pos);
   . . . . .
```

Para determinar cuál de las tres versiones es correcta, suponga que se produce la siguiente secuencia de ejecución:

- P cede el procesador a Q, Q a R, y R completa su ejecución cediendo el procesador a P.

Determine la pila del núcleo de qué proceso se liberaría en cada versión, averiguando de esta forma qué versión es correcta.

Solución

a) El código presenta un error habitual en la programación dentro del sistema operativo: el intento de acceder al mapa de memoria de usuario de un determinado proceso cuando se está ejecutando en modo núcleo dentro del contexto de otro proceso. En este caso, cuando un proceso invoca `_exit` estando el proceso padre del mismo bloqueado esperando en la llamada `wait`, en la siguiente línea de código, se copia el valor de terminación del proceso en su propio mapa de memoria de usuario, en ver de hacerlo en el del padre.

```
*(ppadre->dirstat) = pactual->valexit;
```

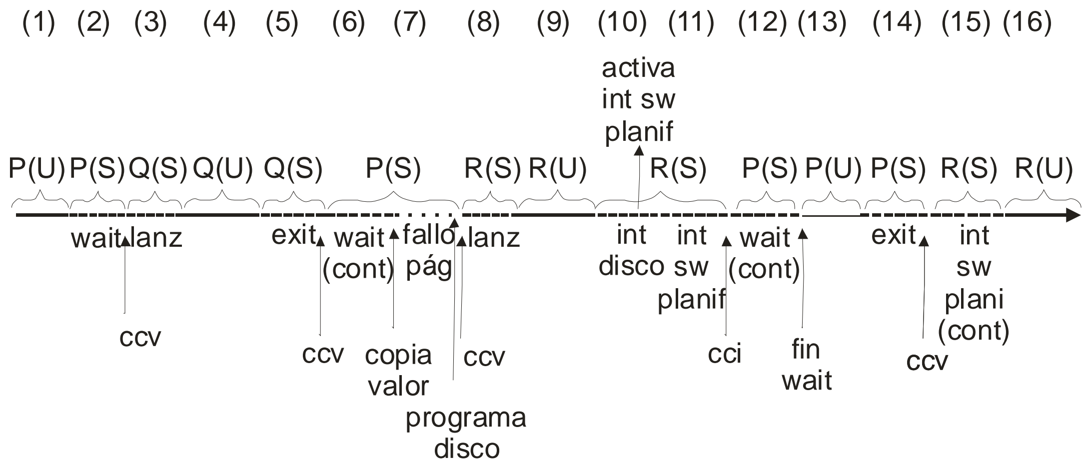
Para arreglarlo, habría que reorganizar el código de manera que la copia del valor de terminación la haga siempre el padre en el `wait`, con independencia de si ya hay un hijo terminado en ese instante o tiene que esperar por el mismo. El código resultante sería:

```
int wait(void *pstatus){
   BCP *phijo_zom; int pidh;
   if (pactual->nhijos==0) return -1;
   if (pactual->nzombis==0)
      bloquear(&(pactual->cola_espera_hijo));

   phijo_zom = obtener_hijo_zombi(pactual);
   *(pstatus) = phijo_zom->valexit;
   pactual->nhijos--;
   pactual->nzombis--;
   pidh = phijo_zom->pid;
   liberar_pila_nucleo(phijo_zom->pila_nuc);
   liberar_BCP(phijo_zom);
   return pidh;
}
```

```
void _exit(int status) {
   BCP *ppadre = pactual->padre;
   liberar_recursos(pactual); //ficheros,mapa,…
   . . . . . . . . . . . . .
   pactual->valexit=(status&0xff)<<8;
   pactual->estado=ZOMBI;
   ppadre->nzombis++;
   if (!vacia(&(ppadre->cola_espera_hijo)))
      desbloquear(&(ppadre->cola_espera_hijo));
   // típico (y correcto) cambio proceso por fin
   . . . . . . . . . . . . // ops. típicas
   pactual = planificador();
   cambio_contexto(NULL, &(pactual->ctx));
}
```

b) La siguiente gráfica muestra la traza de ejecución del ejemplo planteado:



A continuación, se detallan cada uno de los eventos que ocurren durante la traza:

1. **P** en modo usuario realiza una llamada al sistema `wait`.
2. **P** en modo sistema se bloquea al no tener ningún proceso hijo terminado en ese instante, produciéndose un cambio de contexto voluntario a **Q**.
3. **Q** comienza ejecutando en modo sistema una función de lanzadera que rápidamente se completa pasando a ejecutar en modo usuario la rutina inicial del programa.
4. **Q** ejecuta en modo usuario hasta que realiza la llamada al sistema `_exit`.
5. **Q** en modo sistema desbloquea al proceso padre (**P**), que le estaba esperando y, dado que el proceso desbloqueado es más prioritario que el actual, activa una interrupción software expulsiva de planificación dirigida a sí mismo para llevar a cabo el cambio de contexto involuntario de **Q** a **P** cuando complete la llamada en curso (lo que no va a ocurrir al tratarse de la llamada que termina la ejecución del proceso). Esa interrupción software de proceso dirigida al propio proceso **Q** se desactiva, no requiriendo ningún tratamiento (por eso no aparece en la traza), cuando éste completa su ejecución, produciéndose un cambio de contexto voluntario a **P**.
6. **P** en modo sistema continúa la llamada `wait` y pasa a copiar el valor de terminación a la zona del mapa de usuario referenciada por el parámetro de la llamada, pero, dado que la página que contiene esa zona no está residente, se produce un fallo de página que se anida con la llamada al sistema.
7. Al tratarse de una variable global con valor inicial, estará incluida en una región con soporte, por lo que requerirá acceder a disco para traerla (concretamente, al fichero ejecutable puesto que es la primera vez que se accede a esa página). La rutina de tratamiento programa el disco y realiza un cambio voluntario al proceso **R**.
8. **R** comienza ejecutando en modo sistema una función de lanzadera que rápidamente se completa pasando a ejecutar en modo usuario la rutina inicial del programa.
9. **R** ejecuta en modo usuario hasta que llega la interrupción del disco.

10. La rutina de interrupción del disco desbloquea al proceso **P**. Dado que el proceso desbloqueado es más prioritario que el actual, se activa una interrupción software expulsiva de planificación para llevar a cabo el cambio de contexto involuntario de **R** a **P**.
11. La rutina de interrupción software de planificación lleva a cabo el cambio de contexto involuntario requerido, con lo que **P** continúa la ejecución de la llamada al sistema justo en el punto donde quedó detenida.
12. **P** en modo sistema completa la ejecución de la llamada y retornando **P** a modo usuario.
13. **P** ejecuta en modo usuario hasta que realiza la llamada al sistema para completar su ejecución.
14. La llamada de terminación completa la ejecución de **P**, produciéndose un cambio de contexto voluntario a **R**.
15. **R** reanuda su ejecución en el contexto de una rutina de tratamiento de una interrupción software de planificación donde quedó detenida su ejecución. Al completarla, retorna a modo usuario.
16. **R** prosigue su ejecución en modo usuario.

c) Se va a estudiar en cada versión a qué valor hace referencia la variable pasada como parámetro a la operación de liberar la pila del núcleo en el momento en el que P vuelve a ejecutar. Nótese que la situación es muy diferente dependiendo de si la variable es local o global:

- Si la variable es local, el valor que tendrá cuando vuelva a ejecutar P será el mismo que tenía la última vez que ejecutó ese proceso.
- Si la variable es global, el valor que tendrá cuando vuelva a ejecutar P se corresponderá con el valor que escribió en la misma el último proceso que la actualizó.

Primera versión

La variable `prev` es global por lo que su valor se corresponderá con el que escribió R sobre la misma antes de hacer el cambio de contexto a P. Por tanto, hará referencia al BCP del proceso R y liberará **correctamente** la pila de núcleo de ese proceso, que acaba de terminar.

Segunda versión

La variable `prev` es local por lo que su valor se corresponderá con el que escribió P sobre la misma cuando ejecutó previamente. Por tanto, hará referencia al BCP del propio proceso P, lo que es **erróneo**, y dado que ese proceso no ha terminado, no se liberará ninguna pila de núcleo, quedándose la de R sin liberar.

Tercera versión

La variable `pos` es local por lo que su valor se corresponderá con el que escribió P sobre la misma cuando ejecutó previamente. En consecuencia, hará referencia **erróneamente** al BCP del proceso Q, que no es el previsto y que incluso puede que ya no exista y, por tanto, la variable haga referencia a una zona de memoria cuyo contenido es impredecible, quedándose, además, sin liberar la pila de R.