PROGRAMACIÓN DE SISTEMAS

Unidad 5 – Procesamiento en Paralelo

Procesamiento en Paralelo

Tipos de Concurrencia

- Procesos
- Multiplexación E/S
- Hilos

PROCESOS

Procesos

Proceso

Un proceso es una instancia de un programa en ejecución.

Los procesos forma una jerarquía

- Todo proceso tiene una identificación única: el ***process ID***.
- Proceso con ID 0 es el proceso *swapper*.
- Proceso con ID 1 es el proceso *init*

Procesos

- **Identificadores de procesos**

Tenemos varias llamadas que nos devuelven diversos tipos de ID de los procesos.

```
#include <unistd.h>

pid_t getpid( void);

pid_t getppid( void);
```

Procesos

Fork

En Linux, todo proceso “nace” de otro proceso. Un proceso puede crear un nuevo proceso llamando a `fork`:

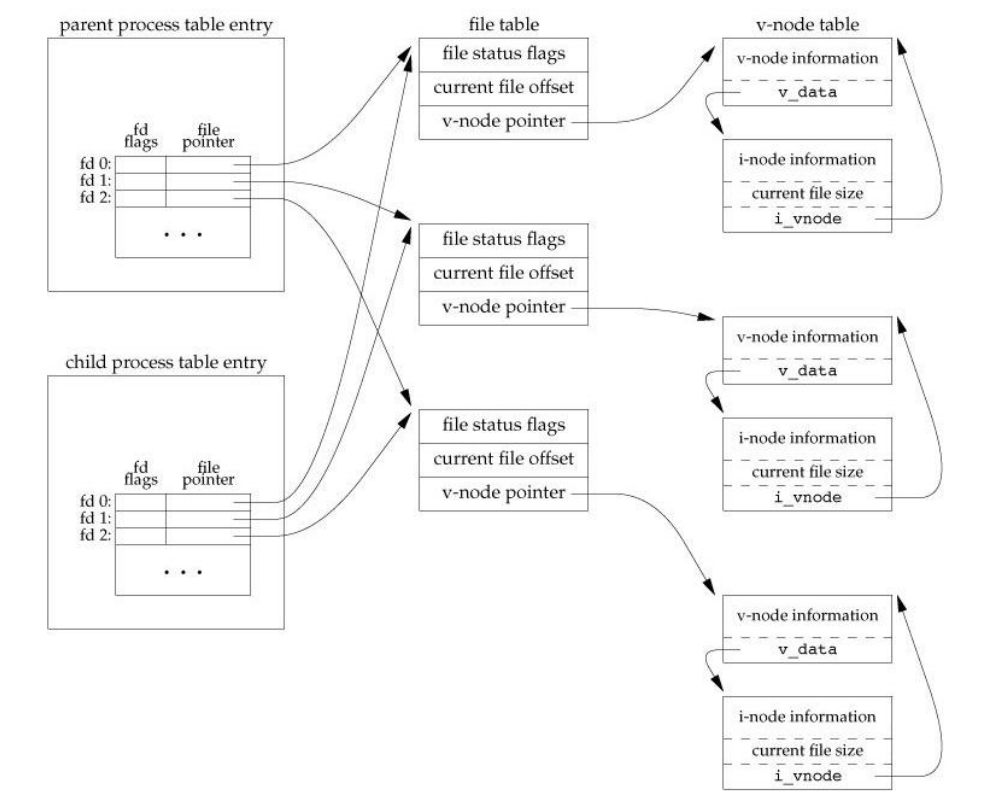
```
#include <unistd.h>

pid_t fork( void);
```

- El nuevo proceso se vuelve el “hijo” del proceso que llamo a `fork`.
- Esta función retorna dos veces (?)

```c
#include "apue.h"
int globvar = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int main( void) {
        int var; /* automatic variable on the stack */
        pid_t pid;
        var = 88;

        if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
                err_sys(" write error");

        printf("before fork\n"); /* we don' t flush stdout */

        if (( pid = fork()) < 0) {
                err_sys(" fork error");
        } else if (pid == 0) { /* child */
                globvar++; /* modify variables */
                var++;
        } else {
                sleep( 2); /* parent */
        }
        printf(" pid = %ld, glob = %d, var = %d\ n", (long) getpid(),
                globvar, var);
        exit(0);
}
```

Procesos

- **Compartiendo archivos**



Procesos

¿Qué es diferente en el hijo?

1. *El valor de retorno de fork.*
2. *Process ID*
3. *Parent process ID*
4. *Valores* `tms_utime, tms_stime, tms_cutime y tms_cstime` *(todos en 0).*
5. *Seguros de archivo no son heredados*
6. *Alarmas pendientes son reiniciadas a 0*
7. *Conjunto de señales pendientes está vacío.*

Procesos

Funciones wait y waitpid

Cuando proceso termina normal o anormalmente, kernel envía señal SIGCHLD al proceso padre.

```
#include <sys/wait.h>

pid_t wait( int *statloc);

pid_t waitpid( pid_t pid, int *statloc, int options);
```

Si padre llamo a wait/waitpid, el proceso puede

1. *Bloquearse, si todos sus hijos están corriendo*
2. *Retornar inmediatamente con el status de terminación del hijo (si hijo terminó).*
3. *Retornar con error.*

Procesos

- `wait` bloquea al proceso que llama la función hasta que alguno de los hijos termine
- `waitpid` espera por proceso con `pid` requerido:
  - *pid > 0 → devuelve hijo con dicho pid*
  - *pid < 0 → devuelve hijo cuyo pid sea igual a |pid| (valor absoluto)*
  - *pid == -1 → devuelve el primer hijo que haya terminado.*
  - *pid == 0 → devuelve cualquier hijo cuyo ID de grupo sea igual al del padre.*

- Ambas retornan el status del proceso en `statloc`

```c
#include <sys/wait.h>
#include "apue.h"

int main( void){

        pid_t pid;

        if (( pid = fork()) < 0) {
                err_sys(" fork error");
        } else if (pid == 0) {
                /* first child */
                if (( pid = fork()) < 0)
                        err_sys(" fork error");
                else if (pid > 0)
                        exit( 0);
                /* parent from second fork == first child */
                /*
                 * We' re the second child; our parent becomes init as soon
                 * as our real parent calls exit() in the statement above.
                 * Here' s where we' d continue executing, knowing that when
                 * we' re done, init will reap our status. */
                sleep( 2);
                printf(" second child, parent pid = %ld\ n", (long) getppid());
                exit( 0);
        }
        if(waitpid( pid, NULL, 0) != pid) /* wait for first child */
                err_sys(" waitpid error");
                /*
                 * We' re the parent (the original process); we continue executing,
                 * knowing that we' re not the parent of the second child.
                 */
        exit(0);
}
```

Procesos

Función exec

- *Son siete funciones*
- *Reemplazan el programa del proceso con el nuevo programa.*
- *No cambian el PID.*
- *Una vez llamada, el proceso llama a* `main`.

- Veamos estas funciones.

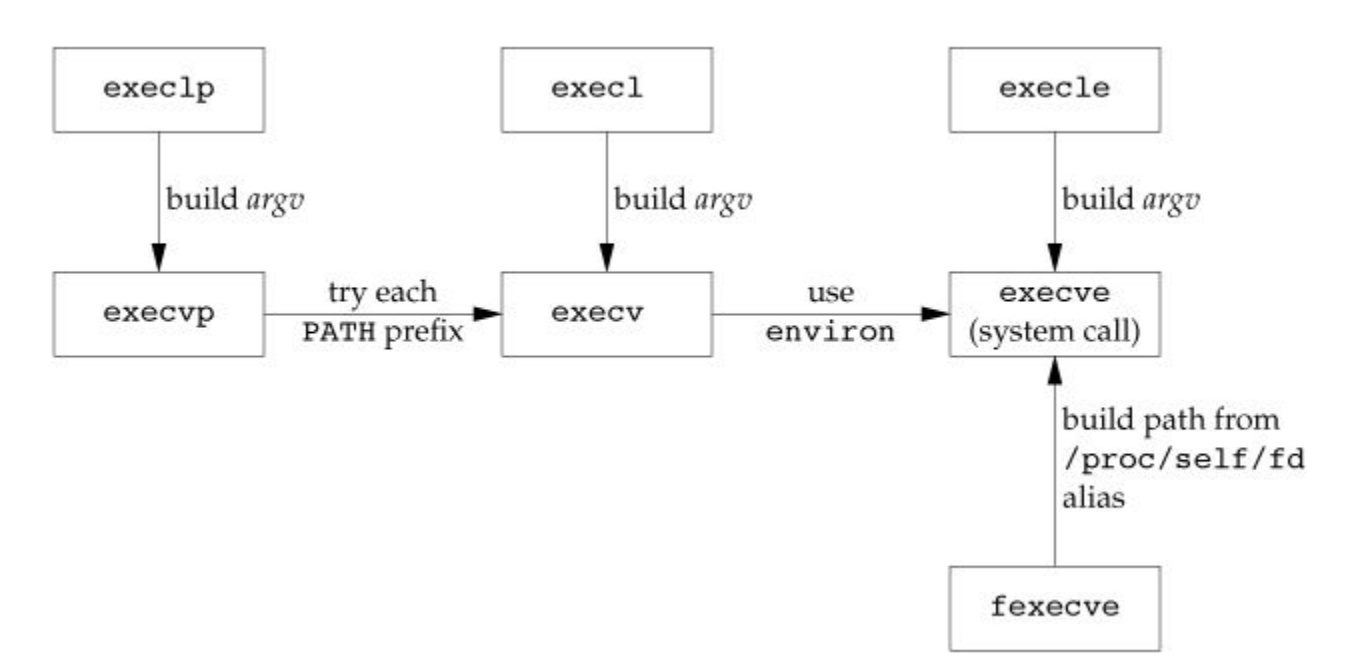
Procesos

```
#include < unistd.h >

int execl( const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv( const char *pathname, char *const argv []);

int execle( const char *pathname, const char *arg0, ... /* (char *) 0 */, char *const
envp[] );

int execve( const char *pathname, char *const argv[], char *const envp[]);

int execlp( const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp( const char *filename, char *const argv[]);

int fexecve( int fd, char *const argv[], char *const envp[]);
```

¡EXEC* NUNCA RETORNA!

Procsos

Relación entre funciones exec

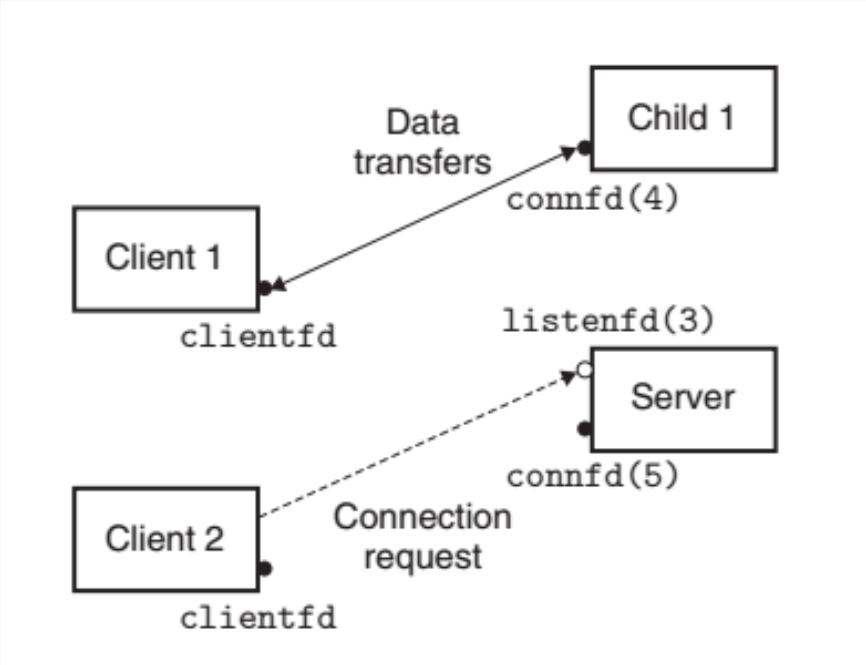


Consideraciones de exec

1. Si `execlp` o `execvp` encuentra el archivo y no es ejecutable, asumen que es un script y trataron de ejecutarlo llamando a /bin/sh
2. `fexecve`, nos despreocupamos de encontrar el archivo, ya quien nos esta llamando nos está dando un descriptor de archivo.
3. `execlp`, `execl` y `execle` necesitan que los argumentos del programa se provean por separado. Ultimo argumento siempre debe ser NULL!
4. Para `execvp`, `execve`, `execv` y `fexecve`, debemos construir un arreglo de punteros a argumentos.
5. Funciones que terminan con e, nos permiten pasarle un puntero a un arreglo de punteros con strings de entorno
6. Las otras copian la variable `environ` del proceso (¿recuerdan setenv y putenv?

Procesos

Paralelizando usando fork



SEÑALES

Señales

Señales

- Las señales son *software interrupts (*interrupciones de software)
- Nos permiten manejar operaciones asíncronas (?)
- En este capítulo estudiaremos:
  - *Conceptos de señales*
  - *Problemas con implementaciones anteriores*
  - *Implementaciones actuales*

Señales

Conceptos de señales

- Toda señal tiene un nombre, empiezan con las letras SIG…
- Ejemplos:
  - SIGABRT se genera cuando proceso llama función `abort()`
  - SIGALRM se genera cuando timer generado por la función `alarm` llega a 0
  - SIGINT se genera cuando usuario presión Ctrl+C
- Las señales estan definidas en `<signal.h>`

Name	Description	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers		•	•	•	•	•	terminate
SIGPOLL	pollable event (poll)				•		•	terminate
SIGPROF	profiling time alarm (setitimer)			•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core

SIGSTKFLT	coprocessor stack fault				•		terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (`setitimer`)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (`setrlimit`)		XSI	•	•	•	•	terminate or terminate+core
SIGXFSZ	file size limit exceeded (`setrlimit`)		XSI	•	•	•	•	terminate or terminate+core
SIGXRES	resource control exceeded						•	ignore

Señales

Generando Señales

1. *Usuario presiona ciertas teclas de terminal (Ctrl+C, etc).*
2. *Excepciones de hardware (division para 0, referencia inválida de memoria).*
3. *Función kill(2) permite mandar señales a otros procesos o grupo de procesos*
4. *Comando kill(1).*
5. *Por condiciones de software (SIGALRM, SIGPIPE cuando pipe no tiene lector, etc).*

Señales

¿Qué puede hacer el proceso al recibir una señal?

1. **Ignorarla**.
    - *Excepto SIGKILL y SIGSTOP. Esto es así para que el kernel para o destruir cualquier proceso.*
2. **Atrapar la señal.**
    1. Kernel llama a una función definida por nosotros (callback), para realizar la acción pertinenete.
3. **Dejar que ocurra acción por defecto.**
    1. Acción por defecto es **terminar el proceso** para la mayoría de las señales

Señales

La función `signal`

```
#include < signal.h >

void (*signal( int signo, void (* func)( int)))( int);
```

- Definida por ISO C.
- `func` puede ser:
  - *La constante SIG_IGN (ignorar)*
  - *La constante SIG_DFL (acción por defecto)*
  - *Una función (un signal handler; a esto le llamamos "atrapar" la señal).*

Señales

- La función que enviamos a `signal` **debe** tomar un entero como argumento, y no retornar nada.
- El valor de retorno de esta función es un puntero al anterior *signal handler*.
- En Linux, esta definición fue simplificada usando `sighandler_t`:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- Veamos un ejemplo.

Señales

Problemas con señales múltiples

- Señales pendientes se bloquean
- Señales pendientes no se encolan
- Llamadas al sistema se pueden interrumpi

Señales

Semántica y terminología de señales confiables

- **Generación de señal:** creada por un *interrupt* de hardware, condición de software, terminal o llamanda a `kill`.
- **Entrega de señal:** se dice señal está entregada cuando se ejecuta la acción asociada a la señal.
- **Señal pendiente:** tiempo entre generación y entrega de señal.
- **Bloqueo de señal:** señal queda pendiente hasta que:
  - *Proceso desbloquee señal*
  - *Cambia acción a ignorar señal.*

Señales

1. Señales relacionadas al proceso son entregadas primero (ej.: SIGSEGV)
2. Cada proceso tiene una **máscara de señales**.
    - *Nos dice las señales que **NO** quiere recibir el proceso.*
    - *1 bit por señal*
    - *Si bit para señal dada es 1, la señal está **bloqueada.***
3. **Máscara** está en una estructura llamada sigset_t, llamado un signal set (conjunto de señales).

Señales

Funciones kill y raise

- Función `kill` envía señal a un proceso o grupo de procesos.

  Función `raise` hace que proceso envíe señal a sí mismo.

```
#include < signal.h >

int kill( pid_t pid, int signo);

int raise( int signo);
```

Señales

Funciones alarm y pause

- Función `alarm` nos permite crear un *timer*, y cuando este expire, enviar la señal SIGALRM. Acción por defecto es terminar el proceso.

```
#include <unistd.h>

unsigned int alarm( unsigned int seconds);
```

- Retorna 0 o número de segundos que faltaron para que expire la alarma anterior.
- `seconds` es el número de segundos a futuro hasta que se genere la alarma.
- ¡Solo se puede tener **una** alarma por proceso a la vez!

```c
#include < signal.h >
#include < unistd.h >

static void sig_alrm( int signo) {
        /* nothing to do, just return to wake up the pause */
}

unsigned int sleep1( unsigned int seconds) {

         if (signal( SIGALRM, sig_alrm) == SIG_ERR)
                return( seconds);
         alarm( seconds); /* start the timer */
        pause(); /* next caught signal wakes us up */
        return( alarm( 0)); /* turn off timer, return unslept time */
}
```

Señales

Signal sets (conjunto de señales)

- Se almacenan en la estructura sigset_t. **Es la representación de las señales.**
- Las manipulamos con 5 funciones:

```
#include < signal.h >

int sigemptyset( sigset_t *set);

int sigfillset( sigset_t *set);

int sigaddset( sigset_t *set, int signo);

int sigdelset( sigset_t *set, int signo);

int sigismember( const sigset_t *set, int signo);
```

Señales

- sigemptyset set inicializa set de tal forma que **TODAS** las señales están **EXCLUIDAS (no serán afectadas por sigprocmask).**
- `sigfillset` inicializa set de tal forma que **TODAS** las señales están **INCLUIDAS (serán afectadas por sigprocmask).**
- Al inicio, una de estas dos funciones **DEBEN** ser llamadas.
- `sigaddset` añade una señal al conjunto set.
- `sigdelset` borra una señal del conjunto
- `sigismember` devuelve 1 si señal esta en el conjunto, 0 si no.

Señales

Sigprocmask

```
#include < signal.h >

int sigprocmask( int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

- Máscara de señales nos dice señales que están bloqueadas para **ser entregadas al proceso.**
  - *Máscara es atributo del proceso*
  - *Podemos cambiarla con la función* `sigprocmask`
  - *Está función afecta solo las señales que están en el conjunto.*

Señales

- `how` nos dice como modificaremos la máscara:
  - *SIG_BLOCK: nueva máscara es **unión** de máscara vieja y set nuevo. Por lo tanto, **set contiene las señales que queremos bloquear.***
  - *SIG_UNBLOCK: nueva máscara es **intersección** de máscara vieja y set nuevo. Por lo tanto, **set contiene las señales que queremos DESbloquear.***
  - *SIG_SETMASK: nueva máscara reemplaza a la máscara vieja**.***

- `oset` es puntero no nulo. Máscara actual es devuelta aquí.
- `set` **es puntero no nulo** de conjunto de señales.

Señales

Función sigaction

- Nos permite **ver** y **modificar** la acción asociada a una señal. Mecanismo más confiable que signal.

```
#include < signal.h >

int sigaction( int signo, const struct sigaction *restrict act,
                          struct sigaction *restrict oact);
```

- Reemplaza a `signal`
- Recibe punteros a dos estructuras del tipo sigaction.

Señales

- La estructura sigaction:

```
struct sigaction {
        void (* sa_handler)( int); /* addr of signal handler, */
                                   /* or SIG_IGN, or SIG_DFL */
        sigset_t sa_mask;          /* additional signals to block */
        int sa_flags;              /* signal options, Figure 10.16 */
        /* alternate handler */
        void (* sa_sigaction)( int, siginfo_t *, void *);
};
```

- `sa_handler` es la dirección de la función que manejará la señal.
- `sa_mask` especifica señales que son añadidas a la mascara de señales del proceso antes de ejecutar el handler.