2. Procesos

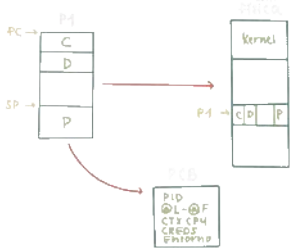
miércoles, 15 de febrero de 2023 11:52

- Un proceso es un **programa en ejecución**
- Las LL.S son servicios
- Cada proceso tiene su propio **espacio de memoria**, que puede estar protegido de otros procesos, y su propio conjunto de recursos asignados, como hilos de ejecución, archivos abiertos, sockets, señales, entre otros. El sistema operativo gestiona los procesos y sus recursos para asegurar la coexistencia pacífica de varios programas en el sistema.
- Cada proceso tiene un **identificador único** que lo distingue de los demás asignado por el propio kernel.
- Cuando ejecutamos un programa:
  - Asignamos memoria al código, datos y pila
  - Iniciamos los registros de la CPU para empezar a ejecutar
  - Accedemos a los dispositivos -> Modo Kernel

2.1. PCB(Process Control Block)

- PCB: estructura de datos de procesos, se almacena en MEM. PRINCIPAL
  - **Gestiona la información del proceso** -> contiene info importante sobre el estado actual del proceso como su id único(PID), estado de ejecución, entorno, recursos asignados al proceso, prioridad del proceso, etc.
  - 1 PCB reservado para cada proceso -> "Tabla" de PCBs
  - El SO accede al PCB para realizar op en un proceso, cambiar su estado de ejecución, asignar recursos adicionales o liberar recursos existentes

Ej: Si creamos un fichero de texto en la terminal -> #gedit p1.txt
Hace una llamada al sistema kernel -> crea un nuevo proceso -> ejecuta el gedit p1.txt -> termina el proceso
Es decir => Crea PCB -> inicia PCB -> reserva memoria -> copia código y datos -> (detalles que dependen del comando) -> libera los recursos -> libera PCB



2.1.1. ESPACIO DE DIRECCIONES(memoria)

- El espacio de direcciones es el rango de memoria a la que un proceso puede acceder durante su ejecución
- Cada proceso tiene su propio espacio de direcciones de memoria aislado de otros y del SO
- Descripción de las regiones de proceso: códigos, datos, pila, montón, …

El PCB contiene información sobre este espació de direcciones como:

1. Su tamaño
2. La dirección base
3. Los permisos de acceso
4. Información de posibles optimizaciones -> seguras
   -> especulativas (SO no sabe si las puede hacer)
5. Qué hacer si hay accesos incorrectos/fallidos
6. …

Esta info es muy importante para el SO para administrar correctamente el espacio de memoria de cada proceso y asi garantizarla protección de la mem.

2.1.2. CONTEXTO DE EJECUCIÓN

- Se refiere al conjunto de datos que describe el estado del procesador y los registros en el momento en el que se interrumpe un proceso y se cambia a otro
- Incluye info sobre el contenido de los registros de la CPU, como el valor del puntero de pila, los registros de propósito general y los registros de estado
- Cuando el sistema operativo cambia de un proceso a otro:
  - Guarda el contexto de ejecución actual del proceso en el PCB correspondiente y carga el contexto de ejecución del siguiente proceso.
  - Cuando el sistema operativo cambia de nuevo al proceso anterior, se carga el contexto de ejecución desde el PCB y se reanuda la ejecución del proceso desde el punto en que se interrumpió.
- PCB y el contexto de ejecución permiten que el sistema operativo gestione múltiples procesos simultáneamente y garantice una transición suave entre ellos

*Si el SO gestiona bien la CPU puede parecer que la máquina tiene más CPUs de las que son-> se ponen en marcha muchos procesos a la vez en la misma CPU, pero la mayoria del tiempo es E/S de datos, así que no hace falta la CPU como tal

2.2. CONCURRENCIA

Capacidad del sistema operativo para gestionar múltiples procesos al mismo tiempo. Esto significa que el sistema operativo puede ejecutar varios procesos en paralelo o de forma simultánea, en lugar de tener que ejecutarlos uno tras otro de manera secuencial.

- Concurrente: no hay ninguna sincronización específica que evite que varios procesos puedan ejecutarse a la vez
- Secuencial: no pueden ejecutarse procesos a la vez porque existen sincronizaciones específicas -> hay dependencia entre procesos

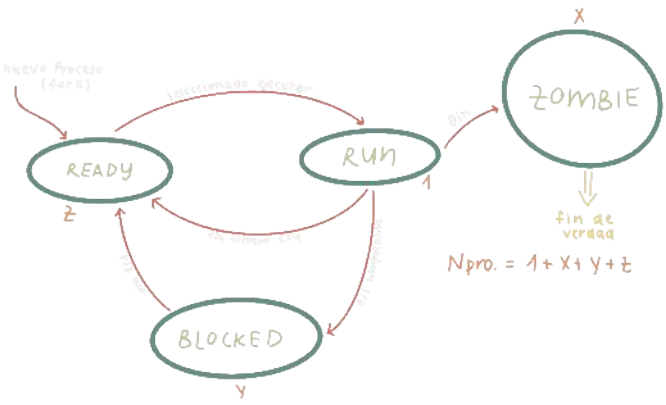
- Concurrencia: capacidad para ser concurrente (misma CPU)
- Paralelismo: varios programas en paralelo, en diferentes CPUs

La concurrencia es importante porque permite a los sistemas operativos aprovechar mejor los recursos del hardware y mejorar el rendimiento global del sistema.
También puede presentar desafíos, como la necesidad de gestionar correctamente la sincronización y la comunicación entre procesos para evitar problemas como la condición de carrera y la inanición.
-> Por lo tanto, los sistemas operativos deben proporcionar mecanismos adecuados para gestionar la concurrencia y garantizar que los procesos se ejecuten de manera segura y eficiente.

2.3. THREADS

2.3.1. ESTADOS DE UN PROCESO(1 thread)

1. RUN -> el proceso tiene asignada una CPU y está ejecutándose
2. READY -> el proceso está preparado para ejecutarse pero aún no tiene asignada una CPU
3. BLOCKED -> el proceso no gasta/consume CPU, está bloqueado esperando a que finalice una E/S de datos o la llegada de un evento
4. ZOMBIE -> el proceso ha terminado su ejecución pero aún no ha desaparecido del kernel/PCB -> se queda esperando a que se consulten las estadísticas



2.4. SERVICIOS Y FUNCIONALIDADES

SERVICIO	LLAMADA A SISTEMA
Crear proceso	Fork
Cambiar ejecutable	Exec(execlp)
Terminar proceso	Exit
Esperar proceso hijo (bloqueante)	Wait/waitpid
Devuelve el PIC del proceso	Getpid
Devuelve el PID del padre del proceso	Getpid

2.5. CREACIÓN DE PROCESOS

int fork();
En particular, la función fork() crea un nuevo proceso (conocido como "proceso hijo") que es una copia exacta del proceso quellama (conocido como "proceso padre").
-> La función fork() devuelve el valor 0 en el proceso hijo y devuelve el ID del proceso hijo en el proceso padre.
-> si retorna -1 = -1 error (NO Hijo)

Esto permite al proceso padre y al proceso hijo realizar diferentes tareas al mismo tiempo, ejecutando diferentes secciones de código y compartiendo cierta información, según sea necesario.

Cuando un proceso crea otro proceso -> relación jerárquica PADRE-HIJO -> A su vez ambos podrían crear otros procesos -> ÁRBOL DE PROCESOS
- El hijo es una copia del padre, con espacio físico diferente, mismos recursos
- Padre e hijo son CONCURRENTES -> El Padre se deja en RUN y el Hijo en READY
- La memoria del hijo se inicializa con una copia de la mem. del padre (código/datos/pila) => MISMO ESPACIO LÓGICO DIFERENTE ESPACIO FÍSICO!!

LLAMADAS A SISTEMA RELACIONADAS CON PROCESOS

EXECLP
- Se utiliza para reemplazar el proceso actual en ejecución con un nuevo proceso que se especifica mediante un programa ejecutable.
- Cuando llamas con **exelp** el proceso actual en ejecución se reemplazará por el programa especificado. El código restante debajo de la llamada NO SE EJECUTARÁ!!
- El nuevo programa se ejecutará en el espacio de memoria que el proceso original, heredará los archivos abiertos y los id de archivo del proceso original
- Si == -1 significa que hay un error durante la mutación
  - Ej: si queremos ejecutar el programa "p1.c" con varios argumentos lo haríamos así:
    execlp(p1.c, "arg1", "arg2", NULL)
    NULL indica el final de la lista de argumentos.

GETPID
- Retorna el PID del proceso que la ejecuta

WAITPID
- Se utiliza para esperar a que un proceso hijo finalice su ejecución y recuperar su estado de salida.
- Cuando se crea un hijo con la llamada fork() el padre puede utilizar waitpid() -> puedes especificar el hijo usando el PID del proceso hijo como argumento
- Puede ser útil para saber si el proceso hijo ha terminado correctamente o no.
  - Ej: **waitpid(PID, A, B) => waitpid(PID,NULL,0)/waitpid(-1,NULL,0)**
    PID -> si pones un -1 un hijo cualquiera
    A -> si no me importa pongo NULL
    &exit_code: variable donde el kernel nos copiará el valor de finalización
    B -> si pongo un 0: bloqueante -> si el hijo no ha terminado bloquea al padre -> SINCRONIZACIÓN

EXIT
- Se termina para finalizar la ejecución del proceso actual en ejecución y devolver un valor de estado al proceso padre que lollamó

PERROR
- Se utiliza para imprimir un mensaje de error en la salida estándar de error

2.6. COMUNICACIÓN ENTRE PROCESOS

Signals -> Eventos enviados(notificaciones) por otros procesos(del mismo usuario) o por el kernel para indicar determinadas condiciones.
Bitmap -> Estructura de datos utilizada por el SO para realizar un seguimiento del estado de los bloques de memoria asignados y cuales están disponibles, también guarda un registro de los signals que estan bloqueados o desbloqueados para cada proceso
Pipes -> Dispositivo que permite comunicar dos procesos que se ejecutan en la misma máquina. Los primeros datos que se envían son los primeros que se reciben. Conectan la salida de un programa con la entrada de otro(Utilizados principalmente por le shell).
FIFOS("First in, First out") -> es una cola de mensajes en la que los procesos pueden escribir y leer datos de manera que se garantiza que los datos se traten en el orden en que se reciben, funciona con pipes con nombre.

2.6.1. TIPOS DE SIGNALS Y TRATAMIENTOS

Para cada evento hay un signal asociado que están predeterminados por el kernel
- El signal es un NÚMERO, pero existen constantes definidas para usarlas en programas o en lineas de comandos

2 signals que no están asociados a ningún evento = PARA QUE LOS USEMOS COMO QUERAMOS -> SIGUSR1 y SIGUSR2

Cada proceso tiene un tratamiento asociado a cada signal
- Tratamientos por defecto
- Capturar(modificar el tratamiento asociado) de todos los signals EXCEPTO -> SIGKILL y SIGSTOP

Nombre	Acción Defecto	Evento
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado
SIGCONT		Continua si estaba parado
SIGSTOP	STOP	Para el proceso
SIGINT	TERMINAR	Proceso interrumpido desde el teclado = Ctr+C
SIGALRM	TERMINAR	El contador definido por la llamada alarm ha terminado
SIGKILL	TERMINAR	Terminar proceso
SIGSEGV	CORE	Referencia inválida a memoria
SIGFPE	TERMINAR	Excepción de punto flotante(división por cero u operación matemática ilegal).
SIGILL	TERMINAR	Intento de ejecutar una instrucción ilegal, código de operación no válido.
SIGUSR1	TERMINAR	Definido por el usuario (proceso)
SIGUSR2	TERMINAR	Definido por el usuario (proceso)

Usos que les daremos principalmente:
1. Sincronización de procesos
2. Control del tiempo (alarmas)

El tratamiento de un signal funciona como una interrupción provocada por software -> al recibir un signal se interrumpe la ejecución del código, pasa a ejecutar el tratamiento que ese tipo de signal tenga asociado y al acabar(si sobrevive) continua desde donde estaba.

Los procesos pueden BLOQUEAR/DESBLOQUEAR la recepción de signals escepto SIGKILL y SIGSTOP (tampoco podemos bloquear los signals SIGFPE, SIGILL y SIGSEGV si son provocados por una excepción).

- Cuando se bloquea un signal -> si se le envía ese signal el proceso no lo recibe y el sistema lo marca como pendiente de tratar. (bitmap asociado al proceso solo recuerda un signal de cada tipo)
- Cuando un proceso desbloquea un signal recibirá y tratará el signal pendiente de ese tipo.

2.6.2. LINUX: INTERFAZ RELACIONADA CON SIGNALS

Servicio	Llamada a sistema
Enviar un signal concreto	Kill
Capturar/repogramar un signal concreto	Sigaction
Bloquear/desbloquar signals	Sigpromask
Esperar HASTA que llega un evento cualquiera(BLOQUEANTE)	Sigsuspend
Programar el envío automatico del signal SIGALRM (alarma)	Alarm

Fichero con signals: /usr/include/bits/signum.h

Hay varios interfaces de gestión de signals incompatibles y con diferentes problemas, Linux implementa el interfaz POSIX

2.6.3. INTERFAZ: ENVIAR/CAPTURAR SIGNALS

- Para enviar:
  Int kill (int pid, int signum)
  - Signum -> SIGUSR1, SIGUSR2, etc.
  - Requerimiento: conocer el PID del proceso destino

- Para capturar un SIGNAL y ejecutar una función cuando llegue:
  Int sigaction (int signum, struct sigaction *tratamiento, struct sigaction *tratamiento_antiguo)
  - Signum -> SIGUSR1, SIGUSR2, etc.
  - Tratamiento: struct sigaction que describe qué hacer al recibir el signal
  - Tratamiento_antiguo : struct sigaction que describe qué se hacía hasta ahora. Este parámetro puede ser NULL si no interesa obtener el tratamiento antiguo.

2.6.4. STRUCT SIGACTION

- SA_HANDLER: es una función que se ejecuta en respuesta a un signal recibido, se encarga de manejar el signal y puede hacer varias cosas como finalizar el proceso, ignorar la señal o realizar alguna función especifica
  - SIG_IGN: ignorar el signal recibido
  - SIG_DFL: usar el tratamiento por defecto
  - Función de usuario con cabecera predefinida: void nombreFuncion(int s);
    IMPORTANTE: la función la invoca el kernel. El parámetro se corresponde con el signal recibido (SIGUSR1, SIGUSR2, etc), así se puede asociar la misma función a varios signals y hacer un tratamiento diferenciado dentro de ella.

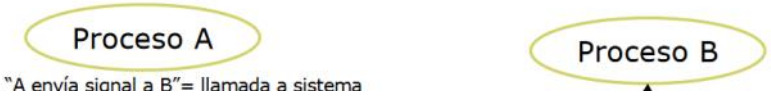
- SA_MASK: conjunto de señales que se bloquean temporalmente mientras se ejecuta el sa_handler de una señal específica, se establece temporalmente para bloquear ciertas señales para evitar que interrumpan la ejecución de handler. Para añadir signals a la máscara de signals que el proceso tiene bloqueados.
  - Si la máscara está vacía solo se añade el signal capturado
  - Al salir del tratamiento de restaura la máscara que había antes

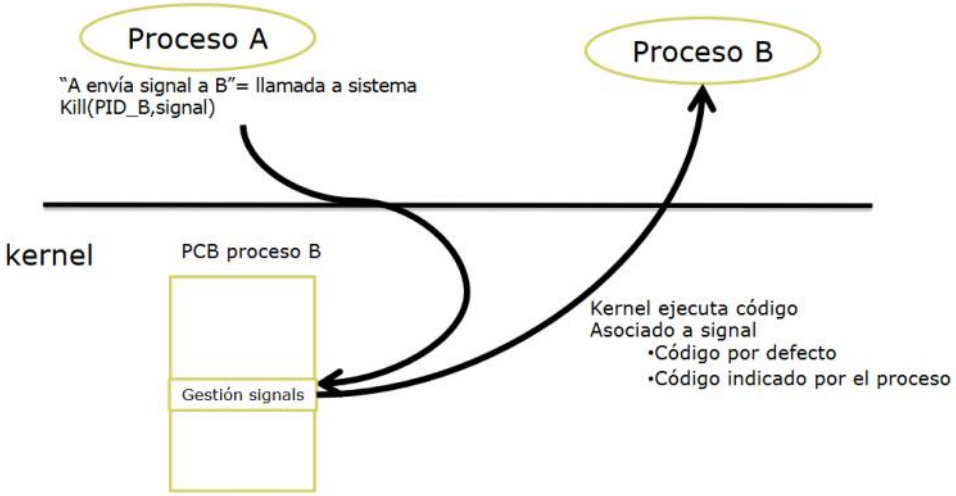
- SA_FLAGS: conjunto de indicadores que se utilizan para configurar el comportamiento del handler(Si = 0 configuración por defecto)
  - SA_RESETHAND: después de tratar el signal se restaura el comportamiento por defecto
  - SA_RESTART: si un proceso bloqueado en una llamada a sistema recibe el signal se reinicia la llamada que lo ha bloqueado

2.6.5. ESTRUCTURAS DE DATOS DEL KERNEL

La gestión de signals es por proceso, la información de gestión está en el PCB
- Cada proceso tiene una tabla de programación de signals(1 entrada por signal) -> Se indica que acción realizar cuando se reciba el evento
- Un bitmap de eventos pendientes(1 bit por signal) -> No es un contador, actúa como un booleano
- Un único temporizador para la alarma -> Si programamos 2 veces la alarma solo queda la última
- Una máscara de bits para indicar qué signals hay que tratar

2.6.6. ENVÍO Y RECEPCIÓN





2.6.7. ACCIONES POSIBLES AL RECIBIR UN SIGNAL