Introducción

Un sistema operativo tiene que ser:

- Usable: claro y útil para el usuario
- Seguro: el sistema no puede caer si tenemos algún fallo (pondrá un mensaje de error)
- Eficiente: cada proceso quiere tener una sensación exclusiva, es decir, que la CPU solo trabaja para él.

Hay dos modos de acceder al Kernel:

- USER MODE: Por ejemplo, cuando hacemos llamamadas al sistema
- KERNEL MODE: Privilegiado. Por ejemplo, cuando ejecutamos llamadas al sistema (síncronas), cuando tenemos excepciones (asíncronas), interrupciones (síncronas), etc.

LLamadas al sistema

Para el programador:

Se hacen desde el código (C, C++, ...). Éste lo envía al kernel para que lo ejecute. Son en USER MODE. Son simples de usar (en principio) Contexto NO modificable (los parámetros de cada función no son modificables) Ej: write(...)

Para el Kernel:

Solo se ejecuta la llamada al sistema, recibida del código. Son en KERNEL MODE. Necesitan que entren parámetros y salgan unos valores. (write -> stdOut)

> Nota: **trap** es el mismo proceso pero de diferente contexto.

Procesos

Conceptos básicos

Proceso: representación del S.O. de un programa en ejecución. Este es nuevo cada vez que ejecutamos.

Programa ejecutándose: tiene código, pila y datos, inicializa los reg de la CPU, da acceso a dispositivos (para el Kernel).

PCB (process control bank): gestiona la información de cada proceso. La PCB contiene 3 partes:

- **Espacio de direcciones:** (1) de código, pila y datos
- **Contexto:** (2) Software: pid, planificación, ppid, ... (3) Hardware: tabla de páginas, ...

Concurrencia: cuando dos procesos son concurrentes es que se ejecutan en paralelo. Aunque haya una sola CPU, hará el efecto virtual, ya que irá dando poco a poco a los procesos lo que necesitan, de forma que parezca que solo "trabaja" para uno.

Estados: un proceso puede tener diferentes estados (en zombie se queda el contxto, NO espacio de direcciones):

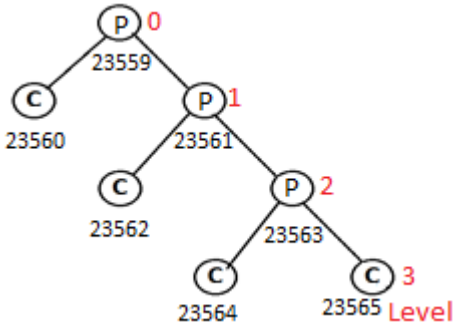
Propiedades del PCB:

- Identidad: PID, credenciales (USERID, GROUPID)
- Entorno: parámetros (argv, HOME, PATH,...)
- Contexto: estado, recursos, ... (durante ejecución)

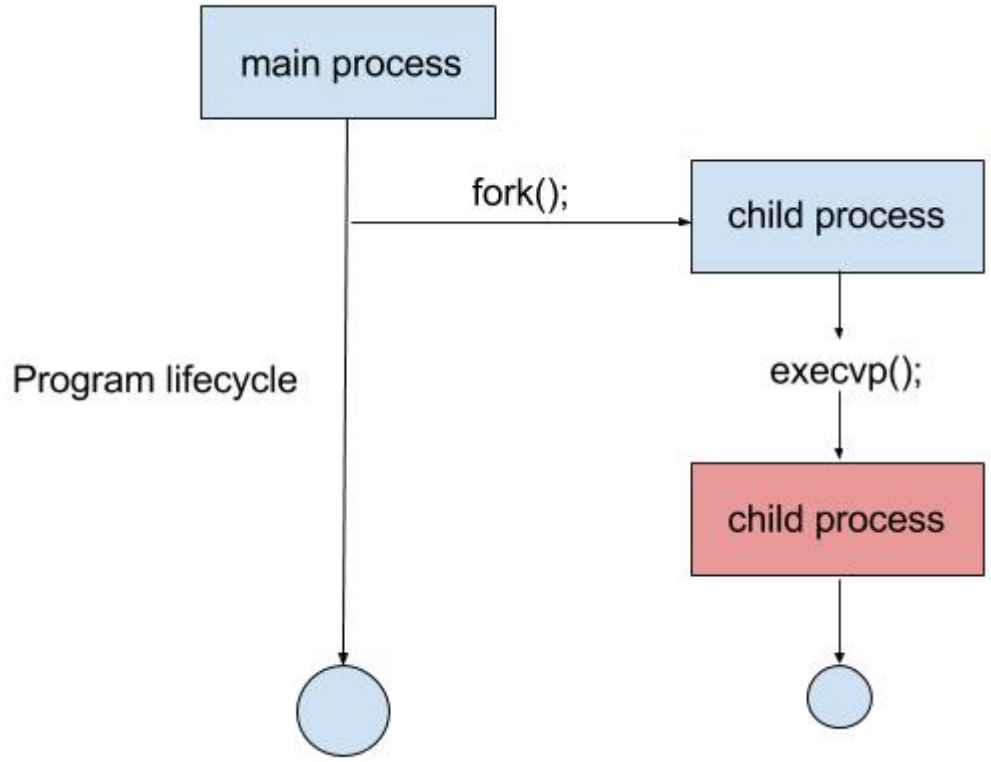
Gestión de procesos

Creación:

fork( ): crea un proceso hijo



execlp( ): cambia espacio de direcciones de un proceso (no contexto)



Cuando un proceso crea a otro, todo se organiza en estructura de árbol.

Cada proceso tiene su PID pero además también tiene el PPID para identificar a su padre.

El sistema operativo decide los recursos que comparte, su planificación y su espacio de direcciones.

Syscall

UNIX --> padre e hijo se ejecutan concurrentemente (a la vez), se duplica el código, la pila y los datos del padre y se le asignan al hijo. - fork -> crea un nuevo proceso (si es el hijo, devuelve 0, si es el padre, devuelve el PID del hijo) - exec -> muta a otro programa (cambia el espacio de direcciones) - exit -> termina el proceso - wait / waitpid -> espera a que muera un proceso (esto bloquea el proceso: RUN -> BLOCKED / WAITING) - getpid -> te da el PID - getppid -> te da el PPID

Hereda

- Código, datos, pila
- **Programación de signals**
- **Máscara de signals**
- Dispositivos virtuales (?)
- userID y groupID
- variables de entorno

NO Hereda

- PID, PPID

- Contadores internos
- **Alarmas y signals pendientes**

Muerte

Si quieres sincronizar el padre con la muerte del hijo haces que el padre espere su muerte con:

```
pid_t pid = fork();
int status = 0;
waitpid(pid, &status, 0);

// Si quieres más info sobre waitpid mira el archivo comandos.pdf
// que hay en la carpeta de Utilidades (SESIÓN 4)
```

Mientras el padre no haga waitpid no se libera el espacio que ocupa el PCB del hijo muerto (ESTADO ZOMBIE). Si el padre muere sin liberar los PCB's de sus hijos el sistema los libera (proceso init).

Mutación

Si queremos mutar a otro programa, se usa el comando execlp.

> Si quieres más info sobre execlp mira el archivo comandos.pdf que hay en la carpeta de Utilidades

Execlp cambia todo el contenido del espacio, pero se mantiene la identidad del proceso.

EJEMPLOS

FORK

```
int ret = fork();

if (ret == 0) {
    // HIJO
}

else if (ret < 0) {
    // ERROR
}

else {
    // PADRE
    // ret == pid del hijo
}
```

ESQUEMA SECUENCIAL

Hasta que el hijo no acaba, el padre no crea otro proceso (porque se queda bloqueado en el waitpid esperando a que su hijo muera).

```
#define NUM_PROCESOS 2
int ret;
for (int i = 0; i < NUM_PROCESOS; i++)
{
    ret = fork();

    if (ret < 0)
        control_error();

    else if (ret == 0) {
        // HIJO
        exit(0);
    }

    else {
        // PADRE
        waitpid(-1, NULL, 0);
    }
}
```

ESQUEMA CONCURRENTE

El padre crea todos los procesos hijos que se ejecutan a la vez y después espera a que acaben todos.

```
#define NUM_PROCESOS 2
int ret;

for (int i = 0; i < NUM_PROCESOS; i++)
{
    ret = fork();

    if (ret < 0)
        control_error();

    else if (ret == 0) {
        // HIJO
        exit(0);
    }
}

while (waitpid(-1, NULL, 0) > 0);
```

CREAR 2^n PROCESOS

```
#define N 10
```

```
for(int i = 0; i < N; i++)
{
    fork();
    printf("Hello World. I'm %d\n", getpid());
}
```

MIRAR EL EXIT_STATUS

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void tratar_exit_code(int status)
{
    if (WIFEXITED(status))
    {
        // Ha terminado por culpa de un exit
        int exitcode = WEXITSTATUS(status);
        printf("Ha terminado por un exit con exit_code: %d\n", exitcode);
    }

    else {
        // Ha terminado por un signal
        int signalcode = WTERMSIG(status);
        printf("Ha terminado con un signal con signal_code: %d\n", signalcode);
    }
}

int main(int argc, char const *argv[]) {
    for (int i = 0; i < 10; ++i)
    {
        // FORK
        int ret = fork();

        // HIJO
        if (ret == 0)
        {
            exit(i);
        }

        // PADRE
        int status;
        waitpid(-1, &status, 0);
        tratar_exit_code(status);
    }
    return 0;
}
```

SIGNALS

Por qué? - Compartir información - Acelerar la computación que realizan - Modularidad

En linux se usan: signals, pipes y FIFOS.

Los signals son notificaciones que informan a un proceso de que ha sucedido un evento.

Evento -> Signal asociado

Están predefinidos por el kernel menos el `SIGUSR1` y el `SIGUSR2`.

Cada proceso tiene un tratamiento asociado a cada signal, pero puedes modificar el tratamiento de cada signal, menos del SIGKILL y SIGSTOP.

SIGNAL	TRATAMIENTO (por defecto)	DESCRIPCIÓN
SIGCHLD	Ignorar	El proceso hijo ha muerto o ha sido parado
SIGCONT	-	Continua si estaba parado
SIGSTOP	Stop	Para el proceso
SIGSEGV	Core	Referencia inválida a memoria (Segmentation Fault)
SIGINT	Terminar	Ctrl + C
SIGALRM	Terminar	La alarma ha sonado
SIGKILL	Terminar	Terminar el proceso
SIGUSR1	Terminar	Definido por el proceso / usuario
SIGUSR2	Terminar	Definido por el proceso / usuario

Funciones

- **kill** -> enviar un signal
- **sigaction** -> reprogramar un signal concreto
- **sigprocmask** -> bloquear signals
- **sigsuspend** -> esperar signals
- **alarm** -> programar la alarma

```
man signal
```

> Si quieres más info sobre estas funciones mira el archivo comandos.pdf que hay en la carpeta de Utilidades

sigaction

```
// Proceso A
int pid = 5555;         // cualquier PID (pid del proceso B)
int signal = SIGUSR1;   // cualquier SIGNAL
kill(pid, signal);      // Le envia el Signal al proceso con pid PID
```

```
// Proceso B
void func(int s) {
    // ...
}
```

```
struct sigaction sa;
/* Inicializar sa con:
    - handler -> func
    - mask -> una máscara que creemos
    - flags -> SA_RESTART
*/
sigaction(SIGUSR1, &sa, NULL);
```

Inicializar un struct sigaction

- **sa_handler**
  - **SIG_IGN** -> ignorar el signal
  - **SIG_DFL** -> tratamiento por defecto
  - **my_func** -> función con cabecera: void my_func(int s)
- **sa_mask**
  - vacía -> sólo se añade el signal que se está capturando
  - Al salir se restaura la anterior
- **sa_flags**
  - **0** -> configuración por defecto
  - **SA_RESETHAND** -> después de tratar el signal se restaura el tratamiento por defecto
  - **SA_RESTART** -> Si estás haciendo una llamada a sistema y recibes un signal, se reinicia la llamada a sistema.

Behind the Scenes

1. El proceso A apunta un signal **en el PCB** del proceso B
2. **El kernel** ejecuta el código (ya sea el por defecto o el que ha especificado el proceso) para tratar el signal de B

Máscaras

Una máscara es una estructura de datos que permite determinar qué signals puede recibir un proceso en un momento determinado de ejecución.

```
sigset_t mask;

sigemptyset(&mask);     // vacía

sigfillset(&mask);      // llena

sigaddset(&mask, SIGNUM)    // añade el signal a la máscara

sigdelset(&mask, SIGNUM)    // elimina el signal de la máscara

sigismember(&mask, SIGNUM)  // devuelve true si el signal está en la mascará, false
si no.
```

Para aplicar una máscara a mi proceso, usamos la función **sigprocmask**.

- **SIG_BLOCK** -> bloquea los signals de la máscara que le pases.
- **SIG_UNBLOCK** -> desbloquea los signals de la máscara que le pases.
- **SIG_SETMASK** -> intercambia las máscaras.

> Para más información mira el archivo comandos.pdf en la carpeta de Utilidades

Para esperar a que llegue un signal puedes usar la función **sigsuspend**, que bloquea el proceso hasta que llega un signal que no es ignorado. Al salir del sigsuspend se restaura la máscara anterior.

> Para más información mira el archivo comandos.pdf en la carpeta de Utilidades

Para sincronizar procesos se puede hacer de dos formas: - **Espera activa** -> si vas a tardar poco - while(!recibido) - **Bloqueo** -> si vas a tardar bastante - sigsuspend

```
void f_alarma() {
    alarma = 1;
}

int main() {
    // Configuramos
    configurar_esperar_alarma();

    // Declaramos
    struct sigaction trat;
    sigset_t mask;

    // Configuramos el sigaction
    sigemptyset(&mask);
    trat.sa_mask = mask;
    trat.sa_handler = f_alarma;

    // Reprogramamos el signal
    sigaction(SIGALRM, &trat, NULL);

    // Esperamos a la alarma (signal)
    alarm(2);
    esperar_alarma();
}
```

Espera activa

```
void configurar_esperar_alarma() {
    alarma = 0;
}

void esperar_alarma() {
    while (alarma != 1);
}
```

Bloqueo

```
void configurar_esperar_alarma() {
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, NULL);
}

void esperar_alarma() {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
    sigsuspend(&mask);
}
```

Gestión interna de procesos

Gestión

- Estructuras de datos: PCB, threads (depende del SO)
- Estructuras de gestión: organizan los PCB en funcion de su estado
  - Normalment colas o listas
  - Eficientes
  - Escalables
- Algoritmos de planificación
- Mecanismos del planificador

PCB

Contiene: - **PID** - **userID** y **groupID** - Estado: **RUN**, **READY**, ... - Espacio para salvar los registros de la CPU - **Gestión de signals** - Información sobre la planificación - Información sobre la gestión de memoria - Información sobre la gestión de E/S - **Accounting (recursos consumidos)**

Planificación

Estructura

Normalmente se organizan en colas o listas. - Cola de procesos - Cola de procesos en **READY** - Cola de procesos esperando **E/S**

El sistema mueve los procesos de una cola a otra.

Políticas

La planificación debe ser muy rápida ya que se ejecuta muchas veces por segundo. Cada 10ms tiene una interrupción de reloj para que un solo proceso no pueda acaparar la CPU.

Un proceso en RUN no puede continuar (necesita datos de E/S, el proceso termina, ...) -> La planificación escoge otro proceso -> Eventos no preemptivos (preemptivo = puede quitar la CPU al proceso, sin que el proceso pueda evitarlo).

- Si le puedo quitar la CPU a un proceso -> EVENTOS PREEMPTIVOS
- Si es el proceso el que voluntariamente deja la CPU -> EVENTOS NO PREEMPTIVOS

Tipos de proceso

Los procesos tienen **ráfagas de computación** y **ráfagas de acceso a E/S** que lo bloquean.

computación > E/S -> **procesos de cálculo** computación < E/S -> **procesos de E/S**

Mecanismos del planificador

Cuando se cambia de un proceso a otro se hace un cambio de contexto que tiene los siguientes pasos:

1. Ejecutando **proceso A**
2. **Guardar Contexto** de A -> PCB de A
3. **Planificador** decide ejecutar B
4. PCB de B -> **Restaurar Contexto** de B
5. Ejecutando **proceso B**
6. Repetir.

Ejecutar código es modo usuario, todo lo demás, es modo kernel.

La planificación decide el **tiempo total de ejecución** de un proceso. Desde que llega al sistema hasta que termina (en todos sus estados) y el **tiempo de espera** del proceso (el tiempo que pasa en ready).

Round Robin

Los procesos **se organizan según su estado**. Están encolados por **orden de llegada**. El proceso recibe la CPU durante **un quantum (10ms ó 100ms)**. El planificador hace una **interrupción de reloj** para que ningún proceso monopolice la CPU.

El RR se activa cuando: - Un proceso **se bloquea** - Un proceso **termina** - Cuando **termina el quantum**

Es un política preemptiva porque puede quitarle la CPU a un proceso.

El proceso que está en RUN -> BLOCKED (cuando acabe irá a READY) El primer proceso en la cola de READY -> RUN

Ningún proceso espera más de (N - 1) * Q milisegundos. Donde N es el número de procesos y Q el tiempo de quantum.

- Q grande -> es como si fuesen **orden secuencial**
- Q pequeño -> produce **overhead** si no es muy grande comparado con el cambio de contexto.

KERNEL

FORK

1. Busca **PCB libre** y lo **reserva**
2. **Inicializar** datos (PID...)
3. Se aplica política de **Gestión de memoria**
4. Se actualizan las estructuras de **Gestión de E/S**
5. (RR) Se añade a la cola de **READY**

EXEC

1. **Código / Datos / Pila** -> NUEVO
2. Se inicializan las **tablas de signals**, contexto, ...
3. Se actualizan las **variables de entorno, argv**, registros, ...

EXIT

1. Se **liberan los recursos** del proceso
2. Se guarda el **estado de finalización** en el PCB
3. Se **elimina de la cola de READY**
4. Se aplica la **política de planificación**

WAITPID

1. Se **busca el proceso en la lista de PCB's** para conseguir su **estado de finalización**
2. **Si está ZOMBIE** -> **el PCB se libera** y se **devuelve el estado de finalización** al padre
3. **Si NO está ZOMBIE** -> el proceso **padre pasa de RUN -> BLOCKED**
4. Se aplica la **política de planificación**

PROTECCIÓN

Niveles de seguridad

1. **Físico** -> Poner las máquinas en habitaciones / edificios seguros.
2. **Humanos** -> Controlar quien accede al sistema
3. **SO** -> evitar que un proceso sature el sistema, asegurar que siempre funcione, asegurar que ciertos puertos de acceso no están operativos, controlar que los procesos no se salgan de su espacio de direcciones.
4. **RED** -> Es el más atacado

Memoria

La CPU sólo puede acceder a la memoria y a los registros, por lo tanto, **las intrucciones y datos deben estar cargados en memoria** para poder referenciarse.

Tipos de memoria

- **Memoria física** -> posición ocupada en memoria
- **Memoria lógica** -> referencia emitida por la CPU

No tienen por qué coincidir si existe un "traducción" (lo veremos más tarde).

- @ del procesador
- @ lógicas de un proceso (el kernel dice si son válidas)
- @ físicas de un proceso (las decide el kernel)

Traducción

@ lógicas <--> @ física

- Cuando se **carga** el programa: El kernel decide donde poner un proceso y se traducen las direcciones cuando las copias en memoria (@ físicas --> @ lógicas).
- Cuando se **ejecuta** el programa: se traduce cada dirección que se genera (@ lógicas --> @ física). Lo hace la MMU + kernel.

La traducción es importante porque facilita la **ejecución concurrente** (en paralelo) de programas cargados en memoria (1 CPU pero N en MF) y facilita el **cambio de contexto** (no necesitamos cargar de nuevo en memoria el proceso).

SO debe garantizar protección: - Cada proceso accede a su memoria, no a la de los demás - MMU detecta accesos ilegales, SO configura la MMU

Asignación de instrucciones y datos se realizan en **tiempo de ejecución**

MMU

Ofrece soporte para la traducción y a la protección. **SO** -> configura la MMU MMU hace **@lógica --> @física** MMU envia excepción si @lógica **no existe** o **no tiene una @física asociada** SO -> gestiona las excepciones de la MMU

Cuando SO cambia la traducción de direcciones?

- Asignar nueva memoria
- Pedir / Liberar memoria
- Cambios de contexto

Protección

- **@lógicas inválidas**
- @lógicas válidas con **acceso incorrecto** (escribir en zona de lectura)
- @lógicas válidas con acceso incorrecto porque **el SO hace alguna optimización**

- COW

En cualquiera de estos casos, la **MMU manda una excepción al SO**

SERVICIOS DEL SO

Carga de programas

Ejecutable DISCO -> Memoria Física

SO: 1. **Lee el ejecutable** 2. **Prepara esquema** del proceso en memoria lógica 1. Estructuras de datos 2. Inicializa la MMU 3. Lee secciones del programa 4. Las **escribe en memoria** 5. **Carga el PC** con el inicio del programa

Formato ejecutable (PASO 1)

La mayoría de sistemas POSIX usan ELF

> Para verlo puedes usar `objdump -h ejecutable`

Esquema (PASO 2)

Cargar

1. Reservar memoria
2. Copiar el binario
3. Actualizar MMU

Optimizaciones

- **Carga bajo demanda**
  - **Una rutina no se carga hasta que se llama.**
  - En las estructuras de datos del SO dicen que la zona de memoria es válida pero en MMU no se le asigna traducción.
  - Cuando el proceso accede a la @lógica, la MMU le dice al SO que no sabe como traducir (excepción). El SO mira en sus estructuras de datos, ve que es una posición válida, provoca una carga y continua la ejecución.
- **Librerías compartidas** y **enlace dinámico**
  - **Los ejecutables no contienen el código de las librerías, sólo un enlace**
  - Se retrasa el enlace hasta el momento de ejecución
  - Ahorra espacio en memoria
  - Facilita la actualización de librerías (sin recompilar)
  - **El binario contiene un stub** un tipo de rutina que hace puente a la que contiene el código

realmente.

Liberar / Reservar memoria

memoria dinámica se almacena en el heap y puedes ajustar su medida a lo que necesites.

```
// Anterior = posición del puntero antes de aumentar el heap
int anterior;
// Tienes que multiplicarlo por el sizeof del elemento que quieras almacenar
int nuevo = 6;
// Nuevo = cuántos elementos quiero almacenar
anterior = sbrk(nuevo * sizeof(int));
```

con sbrk no puedes eliminar una variable que esté en medio del heap

Para ello usamos:

- **malloc**: hace lo mismo que el sbrk pero no siempre aumenta el heap porque no aumenta el tamaño que tu le digas, aumenta más tamaño por si luego necesitas más espacio.
- **free**: no mezclar sbrk's con malloc / free

Errores comunes

```
// ...
for (int i = 0; i < 10; i++)
    ptr = malloc(SIZE);

// Uso de la Memoria
// ...

for (int i = 0; i < 10; i++)
    free(ptr);
// ...
```

En la segunda iteración del bucle fallará puesto que si hemos vaciado ptr, ptr será NULL y por lo tanto no podemos hacer free(NULL).

```
int *x;
int *ptr;

// ...
ptr = malloc(SIZE);
// ...
x = ptr;
// ...
free(ptr);

sprintf(buffer, "%d", *x);
```

Aquí tienes dos punteros que apuntan al mismo sitio por lo tanto si liberas la memoria de uno, el otro apuntará a una posición de memoria no válida y producirá error.

Fragmentación

Cuando la memoria está libre pero no se puede usar para un proceso, tenemos un caso de fragmentación.

- **Fragmentación interna**: memoria reservada para un proceso pero no la usa.
- **Fragmentación externa**: memoria libre pero no puede ser asignada porque no es contigua. Se puede evitar compactando la memoria sin dejar huecos (costoso en tiempo).

Asignación

- **Contigua** -> todo el proceso ocupa una partición que **se selecciona en el momento de carga**.
- **No Contigua** -> aumenta la flexibilidad, la granularidad de la gestión de memoria y la complejidad del SO y la MMU.
- **Paginación**: particiones fijas
- **Segmentación**: particiones variables

Paginación

Esquema

- Espacio de @ lógicas **dividido en particiones de tamaño fijo**.
- **tamaño de las páginas -> marco**
- **Asignar** cada página del proceso en su marco libre -> cuando acaba el proceso **devolver** los marcos asignados a la lista de libres.
- Facilita la **carga bajo demanda**
- Permite **protección a nivel de página**
- **Compartición de memoria** entre procesos
- Una página pertenece a una región de memoria `(código / datos / heap / pila)`

MMU

- Contiene la **tabla de páginas de cada proceso**
  - Validez, permisos de accesos, ...
  - Una entrada por página
- Suele guardarse en memoria
- El SO sabe la @ base de la tabla siempre (PCB)
- TLB: Caché de acceso más rápido
  - Contiene info de traducción de las páginas más activas
  - Cuando cambia la MMU se tiene que invalidar / actualizar la TLB

Problemas

El problema de las tablas de páginas es su tamaño ya que están guardadas en memoria física. Tiene que ser una **potencia de 2**. Normalmente **4Kb**. A medida que crece el espacio lógico de direcciones, se añaden secciones a la tabla de páginas.

Segmentacion

Esquema

- Se divide el espacio lógico **teniendo en cuenta el tipo de contenido**.
- **Aproxima la gestión de memoria** a la visión del usuario
- Tamaño variable -> **segmentos**
- Como **mínimo** un segmento para datos, otro para pila y otro para código

Para cada segmento: 1. **Busca** una partición en la que quepa el segmento 2. Aplica la **política**: - `first fit`(primero que encuentra) - `best fit`(en el que mejor encaje) - `worst fit` (el que más espacio extra tenga) 3. Selecciona la **cantidad de memoria necesaria** para el segmento y el **resto** continúa en la lista de **particiones libres**

Problemas

Puede haber fragmentación externa. No todos los trozos libres son igual de buenos.

Esquemas mixtos

Segmentación paginada

Espacio lógico dividido en segmentos. Segmentos dividios en páginas.

```
size_segmento = num * size_pagina
```

Compartición de memoria

- A nivel de página o segmento:
  - **Librerías** compartidas (implícito)
  - **Memoria compartida** como mecanismo de comunicación

OPTIMIZACIONES

COW

Copy On Write: - **Si no se accede** a una zona nueva -> **no necesitamos reservarla** - **Si no modificamos** una zona que es una copia (cualquier valor después de un fork) -> **no necesitamos duplicar**

Fork

Mientras en el fork no se produzca ningún cambio (no escribamos ninguna variable), se usa el mismo código, datos...

Se van reservando / copiando páginas de memoria a medida que se necesitan.

Cómo?

El kernel asume que no hace falta reservar / copiar la memoria, pero cuando accedemos / modificamos sabe que debe reservar / copiar ya que: - En la **estructura de datos del kernel**: se guardan los **permisos reales** - En la **MMU**: se marcan las regiones con permiso de **solo lectura** - En la **MMU**: pone como **direcciones de lectura** las direcciones físicas asociadas (en el fork pondría que para **leer** acceda al código / datos... del padre, en el caso de reserva de memoria dinámica usa las **páginas comodín**)

Cuando intentas modificar la MMU lanza una excepción y el SO compara sus estructuras con las de la MMU y hace la gestion de la reserva real y reinicia el acceso.

Memoria virtual

- Intenta reducir la cantidad de memoria física asignada a un proceso.
- **Reserva memoria física para la instrucción actual y los datos que esa instrucción referencia**

Ideas

1. Sólo tener **en memoria el proceso activo**:
    1. **Si el proceso necesita más memoria** -> hacer un **swap out** (sacar procesos de memoria)
    2. Tener un dispositivo de **almacenaje secundario (backing storage)** donde se guarden los procesos (con más capacidad que la memoria física)
    3. **Antes de ejecutar** un proceso hay que **cargarlo en memoria** -> Ralentiza la ejecución
2. Evitar expulsar de memoria procesos enteros

Reemplazos

Reemplazo de memoria -> el SO necesita liberar marcos

1. **MMU**: elimina la traducción de la página víctima
2. **Guarda** el contenido de la página víctima en el **área de swap**.
3. **Asigna el espacio (página) que deja libre** la página víctima a la página que necesita estar en memoria

Cuando se intenta acceder a una página que está en el área de swap -> se produce un **fallo de**

página de la MMU.

Mover de SWAP -> Memoria Física: SO necesita:

1. mira que el acceso sea **válido**
2. **Asigna un marco libre** para la página **(reemplaza si es necesario)**
3. **Localiza el área de swap** del contenido
4. La **escribe** en el marco
5. **Actualiza la MMU**

Problemas

- La suma de los espacios lógicos de los procesos puede ser más grande que la memoria física
- Acceder a una página no residente es más lento que acceder a una página residente
  - Porque tiene que hacer la excepción + cargar la página
  - Es importante minimizar el número de fallos de página

Para minimizar el número de fallos se intenta seleccionar "páginas víctima" que ya no se vayan a usar o que tarden más tiempo en necesitarse. (Ejemplo: Least Recently Used)

Se intenta que siempre haya un marco libre.

THRASHING - Invierte más tiempo haciendo el intercambio de memoria que avanzando en la ejecución - No consigue mantener en memoria la cantidad mínima de página para continuar la ejecución - **Detección**: controlar la tasa de fallos de página por proceso - **Tratamiento**: Swap out

Prefetch

Queremos **minimizar el número de fallos** de páginas -> **Anticipamos** las páginas que va a necesitar el proceso y las **cargamos**.

Parámetros: - Distancia: con qué antelación hay que cargar las páginas - Número de páginas a cargar

Algoritmos: - Secuencial - Strided

Resumen (Linux sobre Pentium)

- **exec** -> hace la carga de un nuevo programa
- **fork** -> copia del padre con COW e inicializa la Tabla de Páginas del hijo
- **Planificación** -> se actualiza la tabla de páginas en la MMU en el cambio de contexto y se invalida la TLB
- **exit** -> elimina la tabla de páginas del proceso y libera los marcos.
- **Segmentación Paginada**
  - Tabla de páginas multinivel (2 niveles)
    - Una por proceso, guardadas en memoria y la CPU contiene la @ base.

    - Algoritmo de reemplazo:
        - Aproximación de Least Recently Used
        - Se ejecuta cada cierto tiempo y cuando el número de marcos libres es menor a un umbral
- **COW** a nivel de página
- **Carga bajo demanda**
- **Soporte para librerías compartidas**
- **Prefetch simple (secuencial)**

Jerarquía de almacenamiento