Proyecto Compilando Conocimiento

Ciencias de la Computación

Sistemas Operativos

Una Pequeña (Gran) Introducción

Autores:
Rosas Hernandez Oscar Andrés
Lopez Manriquez Angel
Gonzalez Nuñez Daniel Adrian

Índice general

Parte I

Una Introducción

Capítulo 1

Introducción

1.1. ¿Qué es un Sistema Operativo?

Los sistemas operativos surgen como una solución a la problematica de la **administración de un equipo de computo**, de forma tal que fuese más sencillo trabajar con el y aprovechar al máximo sus recursos.

"Un sistema operativo es un programa encargado de controlar todos lor recursos de una computadora"

1.1.1. Definición Formal y Aburrida

"Un sistema operativo es un software de base compuesto por un conjunto de administradores encargados de la administración (valga la redundancia) de cada uno de los recursos de un equipo de coputo de forma rápida y eficiente."

1.1.2. Características

- **Un sistema operativo es un software de base** debido a que es una plataforma que permite la creación y ejecución de aplicaciones desarrolladas para el propio sistema. Como software de base, el sistem operativo ofrece interfaces para la creación o ejecución de las aplicaciones desarrolladas.

  Incluso mucha gente ve al sistema operativo como una biblioteca estandar, como una base sobre la cual escribir mas software.

- Un sistema operativo esta compuesto de un conjunto de administradores, los cuales controlan todos los recursos del equipo de computo, estos administradores son:

  - Administrador de Procesos
  - Administrador de Memoria
  - Administrador de Entrada y Salida
  - Administrador de Archivos
  - Administrador de Red

- Un sistema operativo debe ejecutarse lo más rapido posible, evitando quitarle tiempo de procesamiento a las aplicaciones de los usuarios, por otro lado debe administrar cada uno de los recursos del equipo de cómputo de forma eficiente, maximizando el uso de cada recurso controlado.

  La rápidez y eficiencia es uno de los principales u objetivos que un Sistema Operativo debe cumplir durante su ejecución.

1.2. Tipo de Sistemas

Sistemas Genéricos:

- Por lotes sencillos
- Por lotes multiprogramados
- De tiempo compartido

Sistemas Especializados:

- Distribuidos: En estos los procesadores no comparten el mismo reloj ni memoria, son sistemas debilmente acomplados
- Paralelos: Los procesadores comparten los recursos como la memoria, son sistemas fuertemente acomplados

1.3. Terminos Básicos aka Cosas que Deberías Saber

- **Spooling:**

  Spool significa "Simultaneous Peripheral Operation On-Line".

  En realidad, lo que sucede aquí es que hay buffer para almacenar los datos, por lo general el disco.

  Los dispositivos de entrada / salida no pueden trabajar a la velocidad de una CPU. Por lo tanto, la salida de la CPU se almacenará en este spool (buffer) y los dispositivos de entrada / salida pueden tomar la salida de este buffer como y cuando se requiera de acuerdo a su velocidad.

  Por lo tanto, la CPU no está vinculada a este dispositivo de entrada / salida y puede realizar otras operaciones. Gracias al spooling podemos mantener la CPU y los dispositivos de entrada y salida trabajando a altas velocidades sin esperar el uno al otro.

- **Reserva de Trabajo:**

  Conjunto de trabajos en el disco listos para ser ejecutados por la CPU

- **Planificación de Trabajo:**

  Es la técnica que se encarga de seleccionar cual será el siguiente trabajo ejecutado en la CPU.

- **Multiprogramación:**

  Técnica utilizada para almacenar múltiples trabajos simultáneamente en la memoria física (RAM).

- **Tiempo Compartido:**

  Técnica utilizada para asignar un tiempo de ejecución a cada proceso lo suficientemente corto para conmutar entre ellos.

  El sistema de tiempo compartido es donde cada proceso se asigna un período de tiempo determinado y el proceso tiene que terminar su finalización dentro de ese lapso de tiempo.

  Si no se logra completar su ejecución, entonces el control de CPU pasa al próximo proceso.

- **Concurrencia:**

  Técnica utilizada ejecutar múltiples trabajos bajo la apariencia de simultaneidad o paralelismo mediante una ejecución secuencial.

- **Memoria Virtual:**

  Técnica utilizada para aumentar o extender la memoria física (RAM) mediante el uso de una pequeña región de disco.

- **Sistema de Archivos:**

  Estructura de almacenamiento de información mediante entes llamados archivos y directorios.

- **Sistemas Paralelos:**

  Sistemas utilizados para el multiprocesamiento compuesto por un conjunto de procesadores que comparten el reljo, memoria y buses del equipo, por lo que se conocen como fuertemente acoplados.

- **Sistemas Distribuidos:**

  istemas utilizados para el multiprocesamiento compuesto por un conjunto de sistemas de cómputo completo que manejan de forma independiente cosas como el reloj, la memoria o los buses. Por esto se le conoce como debilmente acoplados.

1.4. Partes del Sistema Operativo

1.4.1. Vista General

Un Sistema Operativo como cualquier otro software sigue un modelo de ingeniería de software para su diseño y construcción, en modelo que se usa es el denominado por capas, este modelo nos dice que cada capa se encarga de realizar una funcionalidad concreta dentro del sistema operativo.

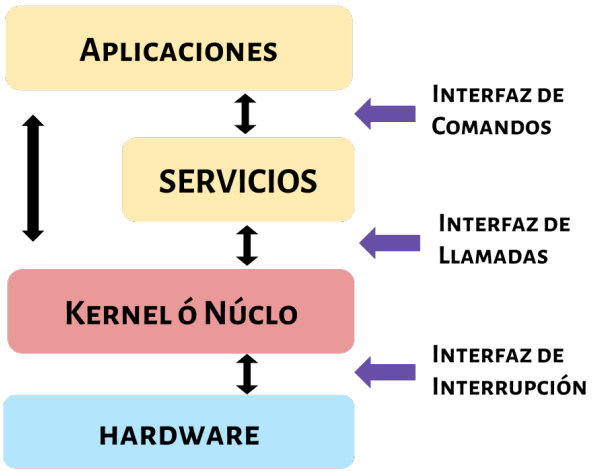
Un sistema operativo normalmente está integrado por las siguientes capas:

- Hardware
- Kernel
- Servicios
- Aplicaciones

Estas capas para llevar a cabo sus funciones requieren comunicarse con cada adyacentes, para lograr esta comunicación se usan las interfaces, estas son:

- Interfaz de Comandos
- Interfaz de Llamafas al Sistema
- Interfaz de Interrupciones

Graficamente las podemos ver como:



1.4.2. Capas

- **Capa de Aplicaciones**

  Esta capa se encarga de mantener cualquier aplicación que el usuario ejecutará en el sistema operativo, siendo la capa con la que el usuario tendrá contacto.

  Se encarga de mantener todas las aplicaciones que tu conoces normalmente.

- **Capa de Servicios**

  Esta capa se encarga de mantener los servicio que apoyan el funcionamiento del sistema operativo, teniendo servicios de seguridad, de mantenimiento, entre otras.

  A veces se suele unir y dividir sus funciones entre la capa de aplicaciones y el kernel.

- **Capa de Kernel o Núcleo**

  Esta es la capa principal del sistema operativo, esta mantiene a los 5 administradores que componen a todo sistema operativo.

- **Capa de Hardware**

  Esta es la capa mas baja del sistema, se encarga de mantener todas las interfaces de comunicación con el hardware.

1.4.3. Interfaces

- **Interfaz de Comandos**

  Esta interfaz comunica a la capa de aplicaciones con la de servicios o a la capa del kernel.

  Esta compuesta de por todos los comandos disponibles en el sistema operativo, siendo la interfaz de interacción inmediata que el usuario posee para comunicarse con el sistema operativo.

- **Interfaz de Llamadas al Sistema**

  Esta interfaz comunica a la capa de servicios con la de aplicaciones o a la capa del kernel.

  Esta compuesta por unas APIs (es decir funciones o métodos) que el sistema operativo pone a dispoción de los usuarios a través de un lenguaje de alto nivel, por esto se le conoce como una comunicación indirecta.

- **Interfaz de Interrupciones**

  Esta interfaz comunica a la capa del kernel con la del hardware.

  Esta compuesta por un conjunto de interrupciones o servicios de interrupción que el sistema operativo pone a dispoción de los usuarios por medio de un lenguaje de programación de bajo nivel, por esto se le conoce como una comunicación indirecta.

1.5. Funcionamiento de un Procesador

1.5.1. Pipeline

Pipelining es una técnica de implementación donde múltiples instrucciones se superponen en la ejecución.

La tubería de la computadora se divide en etapas. Cada etapa completa una parte de una instrucción en paralelo. Las etapas se conectan una a la otra para formar un tubo - las instrucciones entran en un extremo, avanzan por las etapas y salen al otro extremo.

Pipelining no disminuye el tiempo para la ejecución individual de la instrucción. En su lugar, aumenta el rendimiento de la instrucción. El rendimiento de la tubería de instrucciones está determinado por la frecuencia con que una instrucción sale de la tubería.

Debido a que las etapas del tubo están enganchadas juntas, todas las etapas deben estar listas para proceder al mismo tiempo Llamamos el tiempo requerido para mover una instrucción un paso más en la tubería un ciclo de máquina. La longitud del ciclo de la máquina está determinada por el tiempo requerido para la etapa de tubería más lenta.

El objetivo del diseñador de la tubería es equilibrar la longitud de cada etapa de la tubería

Capítulo 2

Kernel de un Sistema Operativo

2.1. Conozcamos a los Administradores

Podemos separar el Kernel en varios administradores, todos interactuan entre si para realizar correctamente su trabajo.

- **Administrador de Procesos:**

  Se encarga de la ejecución de cualquier trabajo en el Sistema, esta compuesto por un conjunto de algoritmos de planificación y de estructuras de datos.

  El hardware con el cual interactua este administrador es el procesador del equipo de cómputo.

- **Administrador de Memoria:**

  Este administrador se encarga de la gestión tanto de la memoria física como de la memoria virtual.

  Esta compuesto por un esquema global de gestión de memoria, así como el conjunto de algoritmos para el control de la memoria.

- **Administrador de Entrada y Salida:**

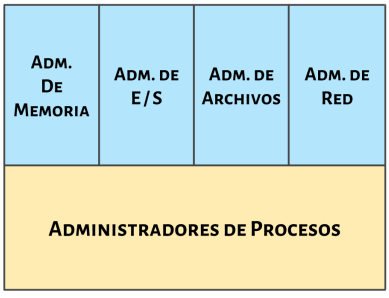
  Este administrador se encarga de la gestión de acceder a cualquier dispositivo externo. Esta compuesto por un conjunto de interfaces de conectividad y controladores de nivel de hardware como software.

- **Administrador de Archivos:**

  Este administrador se encarga de la gestión de todos los archivos del sistema operativo. Se encarga de la organización lógica de la información.

- **Administrador de Red:**

  Este administrador se encarga de cualquier comunicación en red del sistema operativo, esta compuesto por un modelo de comunicación en red y diversos protocolos asociados al modelo usado.



Parte II

Procesos y Virtualización de CPU

Capítulo 3

Procesos

3.1. Introducción

3.1.1. Conozcamos a los Procesos

El Administrador de los Procesos se encarga de la gestión de cualquier trabajo a ejecutar en el sistema.

La clave durante este capitulo será la virtualización, la virtualización es la tecnica que haremos para hacerle creer a cada "programa" que se este ejecutando que es el único en ejecucción y tiene siempre acceso a la CPU.

Haremos esto porque basicamente tenemos una pequeña cantidad de procesadores y muchisimos programas intentando ejecutarse al mismo tiempo.

Ahí esta la clave del problema... ¿Cómo podemos crear la ilusión de tener muchísimas CPU's?

Vamos a introducir una de las abstracciones más fundamentales que nos da el sistema operativo: **Los Procesos**.

El Sistema Operativo logra crear esta ilusión de una gran cantidad de procesadores gracias a la virtualización de la CPU, es decir, gracias a estar corriendo un proceso y luego parandolo para correr el siguiente, esta tecnica es conocida como **tiempo compartido**.

Pata implementar la virtualización y para implementarla bien veremos sobre algunas de los mecanismos a bajo nivel como **el cambio de contexto** o técnicas de alto nivel como **las políticas de planificación**.

3.1.2. Definición de un Proceso

Aquí entre amigos podemos definir a un proceso como un programa en ejecución, así de simple.

Podemos diferenciar un programa en ejecución de otro por el PC (Program Counter), es decir, por cuales instrucciones se estan ejecutando, por el valor actual de los registros, la pila de punteros o cosas así, así que si vamos a asbtraer / generalizar un proceso tendremos que tener una forma de almacenar dicha información que identifica a nuestros programas.

Formalmente podemos decir que: "Un proceso es un programa en ejecución, es la entidad mínima de software ejecutándose en el sistema operativo contenido en una estructura de información"

De esta definición podemos obtener dos características muy importantes:

- **Es una Entidad de Software**

  Esto implica que tendrá asociado un ciclo de vida. Este ciclo representará las diverdad etapas por las cuales puede transitar un proceso.

  Un proceso en el sistema operativo se implementará a través de una estructura de información, la cual contendrá toda la información necesaria para caracterizar a dicho proceso.

  Esta estructura de información es implementada mediante una estructura de datos en el lenguaje de alto nivel que estemos utilizando.

  A esta estructura se le conoce como **Bloque de Control del Proceso** (PCB).

  Este PCB es el elemento con el que el Administrador de Procesos interacturá durante el ciclo de vida del proceso presente en el sistema operativo.

- **Representa una Estructura de Información**

  Un proceso se implementa a traves de una estructura de información, la cual contendra toda la información necesaria para caracterizar completamente cual proceso representado

  Esta estructura de informacion es implementada mediante una estructura de datos en un lenguaje de programacion, y se le conoce como bloque de control de proceso (PCB). Este bloque es un elemento principal ya que el administrador de procesos va a estar en contacto con el durante el ciclo de vida del procesos.

3.2. Bloque de Control de Proceso (PCB)

La imprementación de un proceso en un sistema operativo se lleva a cabo mediante una estructura de datos, esta es conocida como un PCB. Este mantiene toda la información importante de un proceso, esta nos permite caraterizar un proceso en el sistema operativo.

Cada sistema implementa estas estructuras de manera diferente, pero en general todas contienen un:

- **Apuntador:**

  Mantiene un referencia del PCB en la cola de planificación

- **Identificador:**

  Es un simple entero positivo que se encarga identificar al proceso

- **Contador de Programa:**

  Mantiene el valor del registro del procesador PC ó IP, es decir, la dirección de la siguiente instrucción a ejecuctar por el proceso

- **Registros:**

  Estos datos antienen los valores de los registros del procesador que actualmente tiene el proceso en ejecuciòn.

- **Estado del Proceso:**

  Este dato mantiene el estado del ciclo de vida actual, en el que se encuentra la ejecuciòn de un proceso.

- **Información de Planificación:**

  Estos datos mantienen información relacionada a la forma en que un proceso será planificado, entre esta informaciòn se encuentra la siguiente:

  - Algoritmo de planificacion utilizado
  - Prioridad del proceso (en caso de ser utilizada)

- **Información de Memoria:**

  Estos datos mantienen información relacionada con la administración de memoria usada para la ejecuciòn del progceso, esta información incluye lo siguiente:

  - Páginas o segmentos usasdos por el proceso
  - Tabla de páginas o segmentos utilizados por el proceso

- **Información de Contable:**

  Estos datos mantienen información relacionada con diferentes tiempos consmidos por el proceso, entre estos tiempos se encuentran las siguientes:

  - Tiempo de accesso a memoria
  - Tiempo de acceso a dispositivos de entrada y salida.
  - Tiempo de uso de la CPU

El administrador de procesos basa su operación fundamentalmente en la manipulación de los PCB de los procesos actualmente en ejecución en el sistema operativo, siendo su unidad básica funcional.

3.3. Estado de los Procesos

- **Corriendo:** Creo que sobra decir nada sobre esto, es cuando un proceso esta corriendo en la CPU
- **Listo para Correr:** Cuando se esta en este estado todo esta listo para que el proceso siga corriendo, pero por alguna razón el Sistema Operativo aun no ha decidido correrlo
- **Bloqueado:** Se dice que un proceso esta bloqueado si es que no esta listo para volver a correr por ejemplo si es que pidio abrir un archivo, pero el disco duro aun no responde

3.4. La API de los Procesos

3.4.1. Vista General

Veremos pronto una visión particular de cada llamada el sistema, pero antes te mostraré que es lo que debería venir incluido en cada interfaz con el sistema operativo para manejar procesos.

- **Crear Procesos:** Cualquier sistemas operativo debe tener una forma de crear un proceso, esto sería la llamada usada cuando ejecutas un comando o cuando das doble click en el icono de una app.

  Generalmente un sistema operativo te dará varias maneras en las cuales puedes crear un nuevo proceso, pero en todas lo que ocurre tras banbalinas es basicamente lo mismo:

  El proceso es código-instrucciones que se cargan a memoria, se le va a reservar algo de memoria conocida como el stack (esa memoria que se usa para las variables locales, los parametros de una función ó los valores de retorno), seguramente también se encargará de llenar los tan famosos $argc, argv$, también reservará espacio para el heap, este es usado para alocar memoria dinamicamente, como al llamar al *malloc*(), también se encargará seguramente de abrir 3 archivos, uno para errores y una entrada y salida estandar. Finalmente ahora si coloca el apuntador a la instrucción actual al inicio de *main*() por lo que el proceso-programa ahora tiene el control de la CPU.

- **Destruir Procesos:** Cualquier sistemas operativo debe tener una forma de matar un proceso, esto sería la llamada usada cuando nuestro pequeño procesos hace algo que no debe como tomar memoria que no es suya o cuando un proceso bonito finalizo sin mas problemas.

- **Esperar Procesos:** Muchas veces será útil tener una forma de tomar un proceso en ejecución y pasarlo a un estado de "idle" en el que podemos poner en ejecución otro proceso.

- **Información del Procesos:** Muchas veces será útil tener una forma de saber la información personal de cada proceso, cuanto tiempo ha usado al CPU, cual es el ID de su proceso padre o en que estado esta actualmente.

3.4.2. Crear Procesos con $fork()$

Es bastante obvio que $fork()$ nos permitirá crear procesos, pero creeme si te digo que es la llamada a sistema más rara que has visto nunca.

Antes que nada, es importante recordar que para el sistema operativo todo proceso tiene un ID, ó PID, este se obtiene muy fácil con *getpid*()

Lo que hará esta llamada al sistema es crear un nuevo proceso, completamente nuevo, pero a imagen y semejanza del proceso actual, un proceso hijo si lo quieres ver así. Todo será igual, excepto algo:

- El proceso padre obtendrá el PID del proceso hijo como valor de retorno del $fork()$
- El proceso hijo obtendrá un 0 como valor de retorno del $fork()$, además de ser la línea de código siguiente desde la cual va a empezar a correr.

Y si somos extrictos, si algo fallará entonces $fork()$ nos regresa un lindo valor negativo.

Ahora otra cosa que ver es que ahora, nuestro resultado no será deterministico porque no sabemos que se ejecutará primero, si el proceso padre o el hijo, ni cuanto tiempo parasá entre la ejecución de uno y de otro.

Ejemplo

```c
/*═══════════════════════════════════════════════════════════════════════
═══════════                FORK SYSTEM CALL                 ═══════════
═══════════════════════════════════════════════════════════════════════
    USE: $ C99 Fork.c && reset && ./a.out
*/

#include <stdio.h>                                          //We will need this
#include <stdlib.h>                                         //We will need this
#include <unistd.h>                                         //We will need this
#include "SuperSimpleErrorHandling.c"                       //My simple code :p


// ═══════════════════════════════════
// ═══════      MAIN FUNCTION      ═══════
// ═══════════════════════════════════
int main(int argc, char *argv[]) {                          //This is fucking main
    printf("Hello World \t\t(PID %i)\n", getpid());         //Show me your ID

    int ChildID = fork();                                   //Now create a new process
    if (ChildID < 0) return ShowError("Error at Fork", 1);  //Go an show it

    if (ChildID == 0)                                       //YOU ARE THE CHILD?
        printf("I am Child \t\t(PID %i)\n", getpid());      //Show me your ID then kid
    else                                                    //YOU ARE THE PARENT
        printf("I'm Parent of %d \t(PID %i)\n", ChildID, getpid()); //Show me your ID then old man

    return 0;
}
```

3.4.3. Esperar a Procesos Hijos con *wait*() y *waitpid*()

Ya que hemos creado varios procesos nos vemos en un problema algo grave: Ya que el orden en el que se ejecutan dichos procesos no depende de nosotros sino de los algoritmos de planeación sobre la CPU no podemos asegurar en ningun momento si el código que estamos escribiendo sobre un proceso se estará ejecutando antes o después de que terminan sus procesos hijos.

Si quisieramos asegurarnos de esto entonces tenemos dos llamadas al sistema muy muy interesantes:

- *wait*(): Espera a todos los procesos hijos
- *waitpid*($ChildXPID$): Espera a proceso con el PID que le pasemos

3.4.4. Crear Procesos son *exec*()

Ya vimos *fork*() que era la manera más rara de crear procesos pues nos permitia crear varias "copias" del mismo programa en diferentes procesos.

Pero no siempre queremos eso, a veces querremos crear un proceso con un programa completamente diferente.

3.4.5. ¿Porque hay tantas Llamadas al sistema del estilo *exec*∗()

Ok, ok, antes de avanzar mas adelante tengo que contarte algo que a mi me tiene muy sorprendido es la gran cantidad de llamadas al sistema que hay con este estilo, son 5: *execl*(), *execle*(), *execlp*(), *execv*(), *execvp*().

Ahora empecemos, con porque tienen esos nombres tan raros:

- **L vs V**: Define la cantidad de parametros que quieres pasarle al nuevo proceso.
  - **L** si es que solo te interesa pasar uno como en *execl*(), *execle*(), *execlp*()
  - **V** si es que solo te interesa pasar es un arreglo de char* como en *execv*(), *execvp*()

  El formato por array es muy util cuando el número de parametros que vas a enviar al proceso es variable.
- **E**: Las versiones con una e al final le permiten, además, pasar un array de char* que son un conjunto de cadenas añadidas al entorno de procesos generado antes de que se inicie el programa ejecutado.

  En realidad es otra forma de pasar parámetros.
- **P**: Las versiones que NO la llevan requieren un path absoluto o relativo hacia donde esta el ejecutable si es que no esta en el directorio de trabajo.

3.4.6. Sustitución de Código

Decimos que *exec*() crea procesos mediante la sustitución de código, es decir:

- El nuevo proceso sustituye y destruye al proceso padre
- La memoria utilizada por el proceso creador es reutilizada por el nuevo proceso

Ejemplo

```c
/*═══════════════════════════════════════════════════════════════
═══════════════          EXEC SYSTEM CALL          ═══════════════
═══════════════════════════════════════════════════════════════
    USE: $ C99 Exec.c && reset && ./a.out
*/

#include <stdio.h>                                  //We will need this
#include <stdlib.h>                                 //We will need this
#include <sys/wait.h>                               //We will need this
#include <unistd.h>                                 //We will need this
#include <string.h>                                 //We will need this
#include "SuperSimpleErrorHandling.c"               //My simple code :p

int main(int argc, char *argv[]) {                  //Fucking main
    printf("Hello World \t\t\t(PID %i)\n", getpid()); //Show me your ID

    int ChildID = fork();                           //Now create a new process
    if (ChildID < 0) ShowErrorAndGo("Error at Fork", 1); //Go an show it

    if (ChildID == 0) {                             // ═══ YOU ARE THE CHILD ═══
        printf("I am Child \t\t\t(PID %i)\n", getpid()); //Show me your ID then kid

        char *Arguments[3];                         //Create an array of strings :D
        Arguments[0] = strdup("wc");                //New program: "wc" (word count)
        Arguments[1] = strdup("Exec.c");            //Argument to wc: file to count
        Arguments[2] = NULL;                        //End of array
        execvp(Arguments[0], Arguments);            //Run wc & 'exit' process

        printf("This shouldn't print out ever :o"); //All this code stop existing
    }
    else {                                          // ═══ YOU ARE THE PARENT ═══
        int WCID = wait(NULL);                      //This return the pid of the
     child
        printf("I'm Parent of %i aka wc-%i", ChildID, WCID); //Show me your ID then old man
        printf("\t(PID %i)\n", getpid());            //Show me your ID then old man
    }
    return 0;
}
```

3.5. Árbol de Procesos

Internamente, un administrador de procesos utiliza una estructura no lineal para organizar todos los procesos actualmente en ejecución, esta estructura no lineal es conocidada como árbol de procesos.

Un árbol de procesos esta formado por un nodo raíz conocido como proceso principal, cada sistema operativo nombra a este proceso principal de forma particular, sin embargo, la funciòn de este proceso es la misma, funciona como la base para la creación de cualquier proceso en el sistema operativo.

El arbol de procesos clasifica los procesos en dos tipos:

- **Proceso de Sistema**

  Los procesos de sistema son aquellos que conforman al sistema operativo, se caracterizan por ejecutarse en un modo nodo del procesador conocido como modo kernel. En el modo kernel de un procesador se tiene permitido el acceso sin algùn tipo de reztrición a cualquier recurso del procesador y del equipo de còmputo.

- **Proceso de Usuario**

  Los procesos de usuario son todos aquellos procesos ejecutados por los usuarios del sistema operativo, se caracterizan por ejecutarse en un modo del procesador conocido como modo usuario.

  En el modo usuario de un procesador se restringe el acceso a recursos cróticos tanto del procesador como del equipo de cómputo.

En el Árbol de Procesos se le asigna un identificador para referenciar a los procesos, este identificador es un valor entero positivo único, que se asigna una vez que se crea el proceso. Cualquier referencia se lleva acabo a través de este indentificador.

Dentro de este Árbol se establece una relación Hijo-Padre entre los procesos, esta relación permite heredar características de un proceso padre a sus hijos, además nos permite mantener una organización centralizada.

Esto nos dice que no existen procesos aislados, todo proceso tiene que estar contenido en el Árbol de Procesos.

Capítulo 4

Planificación de Procesos

4.1. Introducción

Ahora que hemos visto a los procesos y como es que podemos crearlos ahora hay que ver como que podemos crear un algoritmo que nos permita decidir que proceso ejecutar en cada momento.

4.1.1. Metricas de Planificación

Si queremos encontrar el mejor algoritmo para planificar el orden en que ejecutamos los procesos tenemos que decidir cual será la metrica que usaremos para decidir que tan bueno es nuestro algoritmo.

4.1.2. Tipos de Planificación

Consiste en la seleccion de uno o mas procesos presentes en el sistema operativo para asignarles un recurso requerido. Existen 3 tipos:

- **Planificación a Largo Plazo:**

  Selecciona procesos del disco duro para colocarlos en la memoria física:

  - Tiene el tiempo de ejecucion mas lento
  - Presente principalmente en los sistemas operativos por lotes

- **Planificación a Mediano Plazo:**

  Selecciona procesos del la memoria virtual para colocarlos en la memoria física:

  - Mas rápido que la planeación a largo plazo
  - Presente en sistemas operativos con multiprogramacion y tiempo compartido
  - Estos dos tipos de planificacion definen el grado de multiprogramacion que tiene un sistema operativo, es decir el grado de multiprogramacion es el número máximo de procesos que pueden colocarse en la memoria física simúltaneamente

- **Planificación a Corto Plazo:**

  Selecciona procesos para colocarlo directamente el la CPU:

  - Mas rápido que todas
  - Presente en sistemas operativos gracias al despachador

4.1.3. Despachador

Elemento del administrador de procesos que lleva a cabo el cambio de contexto de la CPU, son de tamañno reducido de hasta 256Kb con tiempos de ejecución contado normalmente en nanosegundo.

Decimos que estamos cambiando de contexto cuando se desaloja el proceso que está en el CPU, previo respaldo del estado actual de ejecución del proceso, y colocar otro proceso en la CPU para iniciar o continuar su ejecución.

Características

- Cambia la CPU de modo usuario a modo kernel
- Respalda el contexto del proceso actual en su PCB respectivo
- Desaloja el proceso actual
- Inicia el contexto del proceso a iniciar por medio de su PCB
- Inicia el PC o IP con las siguientes instrucciones a ejecutar
- Cambia la CPU de modo kernel a modo usuario

4.1.4. Algoritmo de Planificación

Los algoritmos permiten la selección de un proceso bajo algun criterio de planificación respectivo a la cola, para asi asignarle algun recurso requerido.

Todos estos algoritmos se aplican a la cola de planificacion, que es donde se agrupa todos los procesos que solicitan el uso de algún recurso.

Tipos

- Primero en Llegar Primero en Servirse (FCFS)
- Primero el Trabajo mas corto (SJF)
- Algoritmo por Prioridad
- Algoritmo por Torno Circular (Round Robin)

4.1.5. Colas de Múltiplos Niveles

Es como se implementan las colas de planificacion en los sistemas operativos, pues estas estan asociada a una subcola, existen dos tipos:

- **Cola de Múltiples Niveles Simple:**

  Cada que un proceso se asigna a una subcola bajo ninguna circustancia puede ser cambiada a otra subcola.

- **Cola de Múltiples Niveles Retroalimentada:**

  Cualquier proceso se asigna a una subcola puede ser cambiada a otra subcola.

Recuerda que cada subcola puede tener un algoritmo propio, además de tener un algoritmo global que nos permita administrar a cola global.

Capítulo 5

Sincronización

5.1. Comunicación entre Procesos

Los procesos desde el punto de vista de la comunicación pueden clasificarse en dos tipos:

- **Procesos Independientes:** Estos son aquellos que no requieren comunicación con otros procesos para completar su ejecución.
- **Procesos Cooperativos:** Estos necesitan la comunicación para poder completar su ejecución

  Estos requieren la ayuda del sistema operativo para poder llevar a cabo la comunicación, esto se hace mediante los procesos conocidos como IPC (Comunicación Inter Procesos).

5.1.1. IPC

- **Tuberías**: Son archivos creados y controlados internamente por el sistema operativo, sirven como buffers en los cuales se almacenan los datos a comunicar. implementan primitivas para la lectura y escritura de datos.

  A final de cuentas es un archivo que crea el sistema operativo, por lo tanto podemos usar las llamadas al sistema que usamos para archivos.

  Los mensajes a comunicar se manejan como mensajes sin formato. Unicamente soportan comunicación unidireccional con una conectividad uno a uno.

- **Memoria Compartida**: Es una región de memoria creada y controlada por el sistema operativo asociada a un proceso, la región de memoria creada se anexa al espacio de direcciones del proceso que utilizará la memoria compartida.

  Se accede a ella mediante un apuntador a esa región. La memoria compartida puede ser accedida por todos los procesos que lo necesiten.

  Por su naturaleza necesitamos tener una forma de sincronizar toda la información.

- **Sockets**: Son parecidos a la tuberías, sin embargo su implementación se apoya en el administrador de red.

  Todos los mensajes tiene un formato estandarizado, ya que se apoyan con el administrador de red, podemos comunicar procesos de manera local y de manera remota.

  El administrador de red es el que se encarga de seleccionar el protocolo que se hará entre los mensajes.

  Recuerda que el socket es bidireccional.

  Debido justo a que tienen un protocolo son más lentos que las tuberías.

Parte III

Hilos y Virtualización de Memoria

Capítulo 6

Abstracción de Memoria

6.1. Espacio de Direcciones

Antes que nada, recuerda, al Sistema Operativo le importa virtualizar cualquier recurso del pc, eso no solo quiere decir que van a virtualizar la CPU, sino también la memoria, así que grabate eso muy bien, pero muy bien, TODA DIRECCIÓN DE MEMORIA DENTRO DE UN PROGRAMA ES UNA DIRECCIÓN VIRTUAL.

Podemos dividir la abstracción que nos da el Sistema Operativo en 3 partes:

- **Zona del Codigo**: Es literal la sección en la que esta el codigo, es aquí por ejemplo a donde van los punteros a funciones.
- **Stack**: Conocido también como pila, es la parte que se encarga de que mantener la información de las cadenas de llamadas a funciones, es la que se usa cuando se crean variables locales.

  Es conocida a veces como la memoria automatica, porque cuando por ejemplo tu declaras una variale en una función el compilador se encarga de reservar espacio para ella y de liberarla cuando se sale de la función.
- **Heap**: Es la que se encarga de las variables que se crean dinamicamente. Por ejemplo de es la memoria que se reserva cuando se llama a *malloc*()

Ve donde esta tu Memoria Virtual

```c
/*
==========          WHERE IS MEMORY          ==========

    USE: $ C99 CodeStackAndHeap.c && reset && ./a.out                */
#include <stdio.h>                                          //We will need this
#include <stdlib.h>                                         //We will need this

int main(int argc, char *argv[]) {                          //Fucking main
    int LocalVariable;                                      //A var in stack
    printf("Location of Code:  %p\n", (void *) main);       //Show direction for code
    printf("Location of Heap:  %p\n", (void *) malloc(1));  //Show direction for heap
    printf("Location of Stack: %p\n", (void *) &LocalVariable); //Show direction for stack

    return 0;                                               //Go little program
}
```

Capítulo 7

Hilos aka Procesos Ligeros

7.1. Introducción

Son procesos ligeros, es una alternativa a las aplicaciones concurrentes.

Un hilo es un componente de un proceso tradicional que se ejecuta en concurrencia con sus proceso creador.

Para el sistema operativo es más facil crear hilos que crear un proceso tradicional, por eso se le llaman procesos ligeros.

- Comparten valores de los recursos asignados a su proceso creador, siendo la memoria (heap) el principal recurso compartido. Nota, dije que comparten el heap, osea es donde se encuentras las variables que se crean mediante malloc o dinamicamente.
- Manejan de manera independiente tanto la pila como el stack
- Dependen de la ejecución de su proceso padre, el cual es un proceso tradicional

7.2. La API de los Hilos

7.2.1. Estructura de Hilos General

Para poder interactuar con el hilo usaremos una estructura $pthread_t$

7.2.2. Creación de Hilos

Ejemplo de la Declaración

```
1 #include <pthread.h>
2 int
3 pthread_create( pthread_t *        ThreadID,
4                 const pthread_attr_t *  Attributes,
5                 void *            (*Function)(void*),
6                 void *            Argument)
```

Veamos cada uno de los parametros:

- $ThreadID$ es un puntero a la estructura del archivo
- $Attributes$ son los atributos que tendrá el hilo, pero de forma general un $NULL$ será mas que suficiente por default
- $Function$ es la función que ejecutará el hilo, que recibe un puntero generico como parametro y regresará un puntero generico como resultado.
- $Argument$ es exactamente el que le vas a pasar como argumento

7.2.3. Esperar Hilos

Si te gustaría esperar que acabe un hilo antes de continuar la ejecucción del codigo basta con hacer algo como:

Ejemplo de la Declaración

```
1 #include <pthread.h>
2 int
3 pthread_join(    pthread_t *       ThreadID,
4                 void **           Value)
```

Veamos cada uno de los parametros:

- $ThreadID$ es un puntero a la estructura del archivo
- $Value$ es un puntero al valor de retorno que tu esperas osea un puntero a un puntero generico