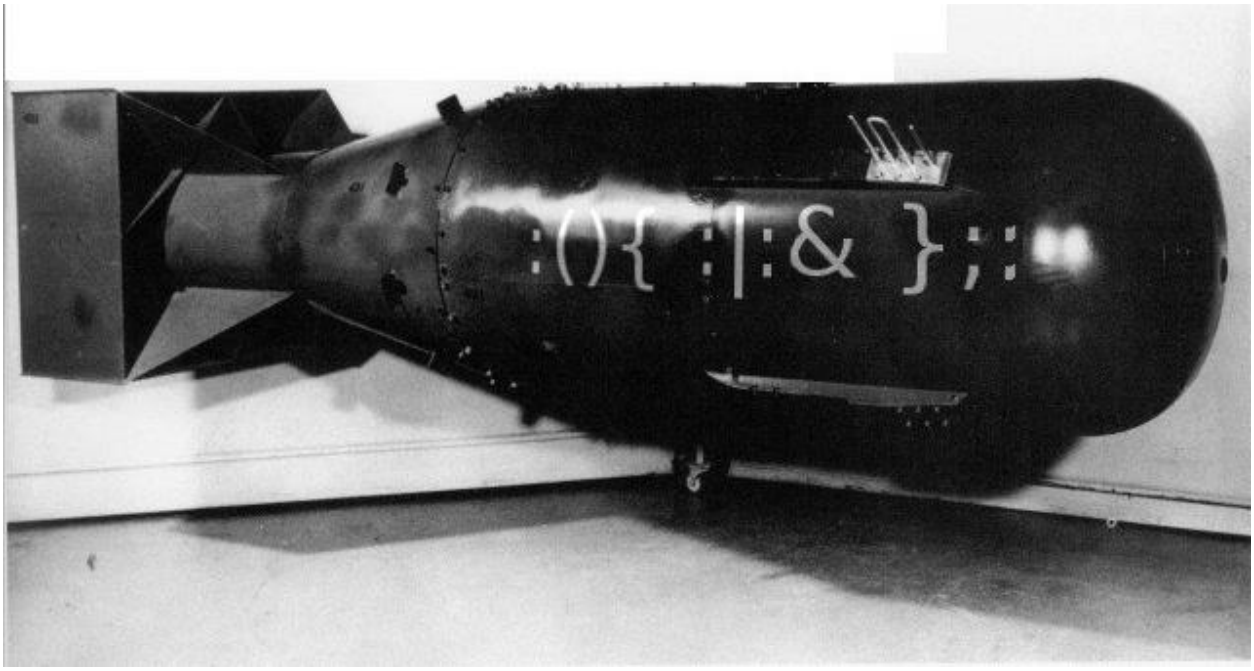
## Procesos

1. Dada esta imagen comentar cual es el chiste y explicarlo en términos del sistema operativo. Dado que se deben utilizar recursos de la web debe ser lo más detallado posible y explicar que sucede, porque y en qué condiciones.




2. En el curso de sistemas operativos hackeamos una computadora que utiliza la NASA. Tenes una única oportunidad para compilar y ejecutar un programa en C utilizando System Call para vulnerar el sistema que harías? Sin programa no se aprueba el ejercicio.

1. Es una fork bomb, y se genera haciendo una gran cantidad de procesos, tantos que se satura, y no se

pueden crear más, a menos que se cierre alguno. Además, los nuevos procesos tmb estan esperando para poder crear más procesos. De esta forma todo se vuelve más lento y tmb inutilizable, ya que no tenemos más memoria disponible y no podemos aprovechar el procesador.

Los paréntesis () definen la función. En su cuerpo, entre las llaves {} recursivamente se llama a si

misma dos veces con un pipeline |, que ejecuta el primer comando enviando su salida al segundo, con el caracter & no se espera a que termine, y los pone en segundo plano de modo que no se puedan terminar... con lo que seguirán consumiendo recursos.

Cada uno de los procesos de la función se llama de nuevo en la función recursivamente y repitiendo el proceso. Como consecuencia el número de procesos del sistema crece de forma exponencial que en poco tiempo termina de agotar los recursos del sistema. A continuación de la función se encuentra el comando: que realiza la primera llamada a la función que desencadena el fork bomb. Esta llamada inicial está separada de la definición de la función por el carácter; que sirve para escribir varios comando, uno a

continuación de otro en una misma línea. El primer comando es la definición de la función y el segundo hace la llamada a la función e inicial el fork bomb.

Este fork bomb crea procesos en el sistema de forma exponencial hasta que el sistema agota los recursos de procesador o memoria en un periodo de tiempo muy corto, menos de unos pocos segundos desde su inicio dada la capacidad de procesamiento de los sistemas actuales. Como consecuencia el sistema requerirá un reinicio.

2.

quiero hacer un proceso hijo y burlarme de la NASA con un chistin de STAR WARS

```
int main () {
   int pid;
   pid = fork();
   if (id == 0) {
      printf("Soy Luck, te destruire, y mi pid es: %d \n", getpid());
      exit(0);
   }
   int status;
   pid_t p = wait(&status);
   printf("Luck, soy tu padre, mi pid es: %d\n", getpid());
   return 0;
}
```

*Corrección*

**NOTA**

Bien menos.

**COMENTARIOS**

el punto 2 dice que solo se puede ejecutar un programa que use syscalls, vas bien el api a usar es la de procesos, pero la idea es ejecutar un excec con "/bin/bash" que te abre la consola del sistema.

---

## Scheduling

### 1. Describa la política Completely Fair Scheduler de linux y relaciónelo con MLFQ. Cual es la idea detras de ambos. Expliquela detalladamente.

En linux el Scheduler se llama Completely Fair Scheduler, y lo que hace es implementar lo que se denomina 'fair-share scheduling' de forma eficiente y escalable. Para lograr estas cosas lo que se intenta es 'gastar' poco tiempo en la toma decisiones de dos posibles formas.

-su diseño.

-uso inteligente de estructuras de datos para la tarea.

Primero que nada, no tiene un determinado time-slice, o sea, no es cte, sino que va APRENDIENDO.

Su funcionamiento:

-Divide de forma justa la CPU entre todos los procesos que están compitiendo por ella => como decía antes, el time slice no es cte.

-Esto se hace con una técnica para contar llamada virtual runtime (Vruntime [ns]):

-virtualizar el tiempo de ejecución -> Vruntime es un run time virtualizado por el número de procesos runnable.

-A medida que un proceso se ejecuta acumula Vruntime. Cuando una decisión de planificación ocurre CFS selecciona al proceso con menos Vruntime para ser el próximo proceso en ejecutarse.

=> Vruntime hace que el proceso sea candidato a ser ejecutado.

¿Cómo sabe el planificador cuando parar de ejecutar el proceso que está siendo ejecutado y correr otro? Acá nos encontramos con un punto de tensión entre performance y equitatividad.

1. Si el CFS switchea de procesos en tiempos relativamente chicos está garantizando que todos los procesos se ejecuten a costa de perdida de performance -> Demasiados context switches.
2. Si CFS switchea pocas veces, la performance del scheduler es buena pero el costo lo tiene la equidad. -> No es justo.

La solución a esto es: Sched_latency.

Sched_latency determina cuánto tiempo un proceso tiene que ejecutarse antes de considerar switcho. (como un timeslice dinámico) -> valor típico 48ns.

CFS divide esta valor por el número de procesos ejecutándose en CPU para determinar el time slice de cada uno de los procesos que hay, y entonces se asegura, por este periodo de tiempo, que CFS va a ser justo.

Ejemplo visto en clase: hay 4 procesos ejecutándose => Sched_latency/4proc = 48ns/4proc = 12ns x proc.

CFS planifica el primer job y lo ejecuta hasta que ha utilizado sus 12ns de virtual runtime, y luego chequea si hay algún otro proceso con menos Vruntime(si lo hay switchea a ese).

Por lo tanto, va cambiando según la cantidad de procesos que hay, cambia con el tiempo.

¿Y si hay muchos procesos ejecutándose? ¿No habría muchos context switches y pequeños time silices? SI.

La solución a esto: min_granularity -> valor típico: 6ns

-CFS nunca cede time-slice de un proceso por debajo de ese número => se asegura de que no haya overhead por context switch.

-nunca le dará menos de ese valor.

Otra solución que se agrega es la interrupción periódica de tiempo para tomar decisiones -> valor típico: 1ns

-en 1ns: tarda en la decisión del prox a ser ejecutado.

Algo que debemos agregar es el cálculo de las prioridades -> Weighting (Niceness)

CFS controla las prioridades de los procesos de forma que los usuarios y admins. puedan asignar + CPU a un determinado proceso

Características:

- cada proceso tiene un valor de prioridad -> por defecto es 0.
- cuanto más chica la prioridad + alto el valor de nice (como la facu)

su cálculo es:

time slice_k = ((weight_k) / suma (weight de todos los procesos)) * sched_latency

También, como nombramos antes, tenemos el uso de una estructura inteligente, que en nuestro caso es el árbol rojo-negro -> árbol balanceado que tiene un O(log n).

Lo que hace es ejecutar el primer nodo (el que tiene menor peso), y sale del árbol para volver a

insertarse, en el caso de que no haya concluido su ejecución, y se inserta en otra posición. Si ya no tiene más unidades de tiempo para su ejecución entonces no se vuelve a insertar en el árbol

Con MLFQ, lo que veo que se puede relacionar es que ambos intentan optimizar de forma eficiente, por ejemplo, MLFQ intenta:

-optimizar el turn arround time mediante la ejecución de la tarea más corta primero.

-que el planificador haga sentir al sistema con un tiempo de respuesta interactivo para los usuarios, por lo tanto, minimiza el response time.

En este caso la solución es utilizando diferentes colas de prioridad para los procesos, pero la idea, en cuanto a mejorar los tiempos y la eficiencia, tienen el mismo objetivo.

*Corrección*

**NOTA**

Regular.

**COMENTARIOS**

el ejercicio es a libro abierto, la parte teórica es lo que está en los apuntes y el núcleo, que es la idea detras de las políticas no esta

---

## Memoria Virtual

1. Describa en pseudocigo o en el lenguaje que quiera como funcionaria un algoritmo de address translation en un modelo de memoria de segmentacion paginada y con tlb de un solo nivel. La v.a. (segmento | page# | offset).

## Concurrencia

2. Un alumno de sisop tiene que rendir el final y no entiende lo que es un deadlock, como prepararía para rendir.

1. MEMORIA VIRTUAL

vpn -> virtual page number de la virtual address

```
int address_trans_tlb (vpn, tlb){
   if (vpn.in(tlb)) {
      //HIT -> la tlb tiene la traducción
      PFN = tlb.getvpn(VPN)  // me devuelve la page frame number de la entrada de la tlb
      offset = tlb.getoffset(PFN) //devuelve el offset
      return paddres = PFN + ofset
   }
   //MISS -> la tlb no tiene la traducción y hay que hacerla.
   page_table_trad = encontrarTrad(vpn)
   if(page_table){
      tlb.settrad(page_table_trad)
      return page_table_trad;
   }
   return -1;
}
```

No tengo tiempo de hacer el cod de la traducción, lo explico rápido:

Tengo una dirección virtual, que se divide en 3 partes |10 bits|10 bits|12 bits|

Los primeros 10 bits: son el índice de la page directory, que contiene la dirección

a la page table.

Los segundos 10 bits: son el índice de esa page table nombrada anteriormente.

Los últimos 12 bits: offset.

Hay solo 1 page directory:

-es la entrada a toda la memoria del proceso

-la dirección del page directory es conocida por el proceso

-tiene 1024 entradas de 4btes -> esas entradas son punteros a una page table

=> hay 1024 page tables.

Tienen una dirección base que va a ser la dirección del frame físico + un IDO al offset = se obtiene la dirección exacta física de donde esta lo que se busco

(la dir base siempre está en el c contexto del proceso en un registro.)

2.CONCURRENCIA

DEADLOCK

---

En la concurrencia hay algunos errores comunes, uno de ellos es el deadlock, o deadlock bugs. Los deadlocks bugs aparecen en concurrencia cuando entre dos o más threads uno obtiene el lock y por algún motivo nunca lo libera, haciendo que los demás se bloqueen.

Para que aparezcan se tienen que dar unas condiciones necesarias, y son las siguientes:

-Exclusión mutua: los threads reclaman control exclusivo sobre un recurso compartido que necesitan

-Hols_and_wait: un thread mantiene un recurso reservado para sí mismo mientras espera que se dé alguna condición. -> lo agarra y espera.

-No preemption: no hay forma de sacarlo, ya que los recursos adquiridos no pueden ser desalojados por la fuerza.

-Circular_wait: hay un conjunto de threads que de forma circular cada uno reserva uno o más recursos compartidos que son requeridos por el siguiente en la cadena. -> todo esperan a que otro haga algo y nadie hace nada...

Una buena forma de entenderlo, si aún no se comprende el tema de deadlock, es con el Caso de los Filósofos, visto en clase.

Hay, en una mesa redonda, cinco filósofos sentados esperando para comer con sus DOS palillos chinos (Exclusión mutua). Cada uno tiene un plato de fideos y un palito a la izq. de su plato. Para comer si o si necesitan ambos palitos, por lo que si tienen uno solo no podrán hacer nada... Cada uno solo puede tomar el de su izq. y derecha, otros no. Además, cada vez que uno agarra un palito NO lo suelta (no preemption) Si todos toman el palito a su izq. al mismo tiempo, lo que se genera es CIRCULAR WAIT, ya que todos están esperando para siempre que otro suelte su palito para poder agarrar dos y comer... pero nunca pasa. (Hols_and_wait, ya que agarran un recurso que otro necesita, pero esperan la condición de tener para poder comer, por lo que no hacen nada, pero le sacan el recurso a los demás).

*Corrección*

**NOTA**

Bien.

## File System

Construya un comando con las siguientes syscalls

```c
#include <sys/types.h>
#include <dirent.h>

struct dirent {
    ino_t d_fileno;               // i-node nr.
    char d_name[MAXNAMLEN + 1];   // file name
}

struct stat {
   dev_t     st_dev;         /* ID of device containing file */
   ino_t     st_ino;         /* Inode number */
   mode_t    st_mode;        /* File type and mode */
   nlink_t   st_nlink;       /* Number of hard links */
   uid_t     st_uid;         /* User ID of owner */
   gid_t     st_gid;         /* Group ID of owner */
   dev_t     st_rdev;        /* Device ID (if special file) */
   off_t     st_size;        /* Total size, in bytes */
   blksize_t st_blksize;     /* Block size for filesystem I/O */
   blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
   time_t    st_atime;   /* time of last access */
   time_t    st_mtime;   /* time of last modification */
   time_t    st_ctime;   /* time of last status change */
};
```

```
DIR * opendir (const char *dirname);
struct dirent * readdir (DIR *dirstream);
int closedir (DIR *dirstream);

int stat(const char *pathname, struct stat *statbuf);
```

Crear un comando linux que reciba una lista de nombres de archivos y para cada archivo sino existe lo cree y si existe cambie la fecha de ultimo acceso a la fecha actual del sistema.

```
#include <time.h>


int main (int argc, char* argv[]) {
   if (argc < 1) {
      printf("Se debe ingresar al menos un parametro.\n");
      _exit(-1);
   }


   char* names[] = argv[0];


   for (int i = 0; i < sizeof(names); i++) {
      DIR * directory = openddir(names[i]);
      struct dirent * ep;
      ep = readdir (directory);


      if(!ep) {
         time_t t;
         time(&t);
         ep->st_mtime = t;
         }
         closedir(directory);
```

```
        }
            else {
           params = S_IRUSR | S_IWUSR;
           if (fd = mkdir(directory, params) < 0) {
              perror("mkdir() error");
           }
           closedir(fd);



        }
     }
}
```

*Corrección*

**NOTA**

Regular.

**COMENTARIOS**

falta iterar y buscar que los nombres estén en la lista.