Programación Avanzada (TC2025)

Tema 3. Administración de procesos

Instituto Tecnológico y de Estudios Superiores de Monterrey. Campus Santa Fe
Departamento de Tecnologías de Información y Electrónica
Dr. Vicente Cubells (vcubells@itesm.mx)

Temario

- Los procesos dentro del SO
- El entorno de un proceso
  - Parámetros
  - Variables de entorno
- Creación, ejecución y terminación de procesos
- Manipulación de las relaciones entre procesos

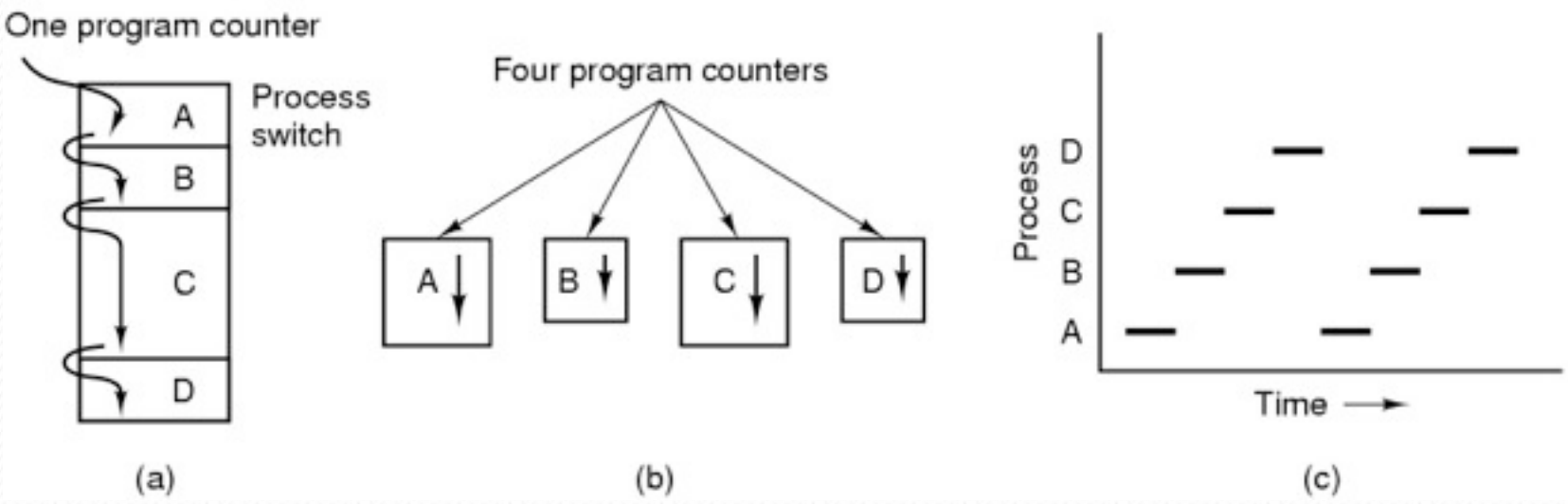
Procesos

¿Cómo ve el sistema operativo a una aplicación?

¿Qué es un proceso?

- De acuerdo a Deitel
  - programa en ejecución, actividad asíncrona
  - “espíritu animado” de un procedimiento
  - la entidad a la que se asignan los procesadores
- De acuerdo a Tanenbaum
  - Un programa en ejecución, incluyendo:
    - Contador del programa, registros y variables
- Conceptualmente, cada proceso tiene su propia CPU virtual

El modelo de procesos



a) Multiprogramación de 4 programas
b) Modelo conceptual de cuatro procesos secuenciales independientes
c) Solo un programa activo en cada instante

Creación de un proceso…

- Cuatro eventos lo causan:
  - Inicio del sistema
    - Foreground
      - Interactúan con el usuario
    - Background (e-mail, web servers)
      - Demonios
  - Ejecución de una llamada *fork()* por otro proceso
    - Creación de procesos hijos
    - Ejemplo: obtener y procesar datos de la red
    - ¿Qué pasaría en un sistema multiprocesador?

Creación de un proceso

- Cuatro eventos lo causan:
  - A solicitud de un usuario
    - Sistemas interactivos
    - Mediante una orden
      - Ejecutando un comando
      - Click del mouse
  - Procesamiento de un trabajo por lotes
    - Cola de trabajos pendientes
- Unix vs. Windows
  - *Fork()* vs. *CreateProcess()*
  - Ambos casos, espacios de memoria separados

Terminación de un proceso

- Cuatro condiciones para terminar:
  - Salida normal (voluntariamente)
    - *Exit(), ExitProcess()*
  - Salida por error (voluntariamente)
    - Compilar un archivo inexistente
  - Error fatal (involuntariamente)
    - División por cero, referencia a memoria inválida
  - Eliminado por otro proceso (involuntariamente)
    - *Kill(), TerminateProcess()*
    - Algunos sistemas eliminan todos los procesos hijos
    - Unix y Windows, no

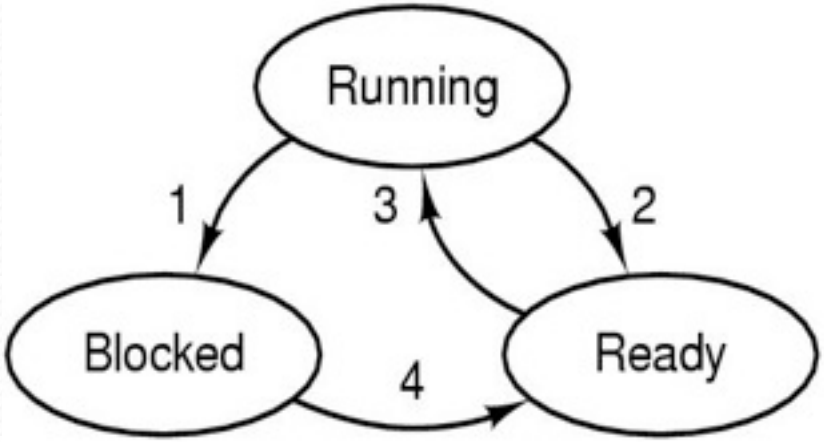
Jerarquías de procesos

- Los procesos pueden crear procesos hijos, estos a su vez pueden crear otros procesos
- Se construye una jerarquía
  - UNIX llama a esto "*process group*“
  - Ejemplo: *init()* y varias consolas
- Windows no maneja el concepto de jerarquía de procesos
  - Todos los procesos son creados por igual

Estados de un proceso…

- Interacción entre procesos
  - Cada proceso es una entidad independiente
  - cat cap1 cap2 cap3 | grep “proceso”
- Un proceso se bloquea por:
  - Espera de la entrada
  - El planificador le quitó la CPU
  - Ambas son diferentes

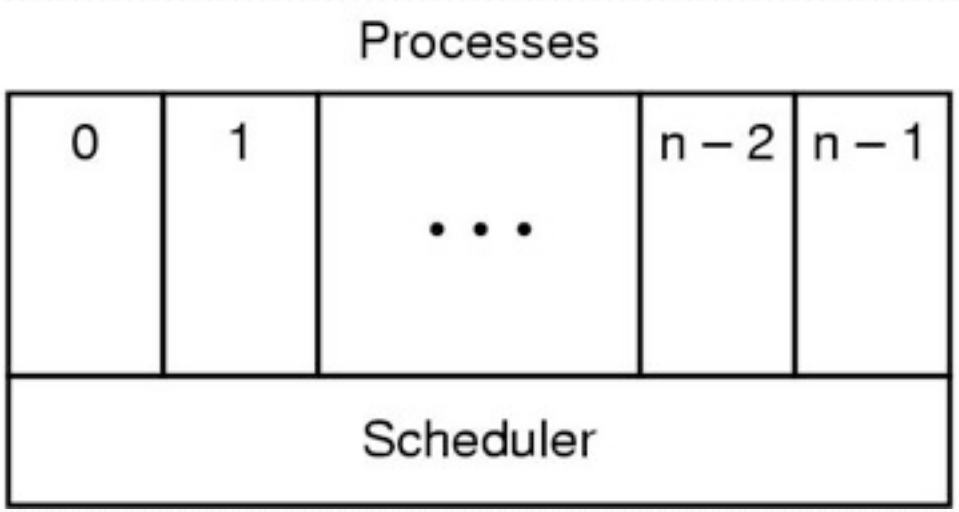
Estados de un proceso…



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Posibles estados de un proceso
  - En ejecución
  - Listo
  - Bloqueado
- Transiciones entre los estados

Estados de un proceso



- El *scheduler* es el nivel inferior de los SO estructurados a procesos
  - Maneja las interrupciones, programación de la CPU
- Encima de este nivel se encuentran los procesos secuenciales

Implementación de procesos…

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Campos de un entrada en la tabla de procesos

Implementación de procesos

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Operaciones que realiza el SO al ocurrir una interrupción

El entorno de un procesos

Procesando argumentos de la línea de comandos

- Procesando argumentos estándares de POSIX
  - Ver ejemplo: `t3c1e1`
- Procesando argumentos largos al estilo GNU
  - Ver ejemplo: `t3c1e2`
- Revisar la función `argp_parse` del header `<argp.h>`

Trabajando con variables de entorno <stdlib.h>

- **char** * getenv (***const char*** *name)
  - Regresa el valor de una variable de entorno
- **int** putenv (***char*** *string)
  - Adiciona o elimina una variable de entorno
- **int** setenv (***const char*** *name, ***const char*** *value, ***int*** replace)
  - Adiciona o reemplaza una variable de entorno
- **int** unsetenv (***const char*** *name)
  - Elimina una variable de entorno
- **int** clearenv (***void***)
  - Elimina todas las variables de entorno
- **char** ** environ
  - Variable de <unistd.h> que almacena un arreglo de cadenas con el formato nombre=valor

Creación de procesos

Ejecutando comandos de Unix desde C <stdlib.h>

- Ahorro de tiempo, claridad en los programas

```
#include <stdlib.h>

main()
{
   printf("Archivos en el directorio son: \n");

   system("ls -l");
}
```

La función `system()` es una llamada que está construida de otras 3 llamadas del sistema: `execl()`, `wait()` y `fork()`

La llamada exec <unistd.h>

- La función `execl()` realiza la ejecución (execute) y salida (leave) de un proceso
- Esta definida como:

```
int execl (const char *filename, const char
*arg0, ...)
```

```
#include <unistd.h>

main()
{
   printf("Archivos en el directorio son: \n");

   execl("/bin/ls","ls","-l", (char *) NULL);

}
```

Otras variantes de exec

<unistd.h>

- **int** execv (***const char** *filename, **char** ***const** argv[])
  - Recibe los argumentos en un arreglo
- **int** execve (***const char** *filename, **char** ***const** argv[], ***char** ***const** env[])
  - Permite establecer el enviroment de ejecución
- **int** execle (***const char** *filename, **const char** *arg0, **char** ***const** env[], **...**)
  - Igual que execl pero permite establecer el enviroment
- **int** execvp (***const char** *filename, **char** ***const** argv[])
  - Busca en el PATH por el comando a ejecutar
- **int** execlp (***const char** *filename, **const char** *arg0, **...**)
  - Como execl pero realiza la misma búsqueda en el PATH

La llamada `fork` `<unistd.h>`

- La función `fork()` cambia un proceso único en 2 procesos idénticos, conocidos como el padre (parent) y el hijo (child)
- En caso de éxito, `fork()` regresa el PID del proceso hijo al proceso padre. En caso de falla, `fork()` regresa al proceso padre -1, pone un valor en `errno`, y no se crea un proceso hijo

```
#include <unistd.h>

main()
{
          int valor = 0;

          printf("Bifurcando el proceso\n");
          valor = fork();
          printf("El id del proceso es %d, el proceso padre es %d  y el valor
regresado es %d\n",
                    getpid(), getppid(), valor);

          execl("/bin/ls","ls","-l",0);
          printf("Esta linea no es impresa\n");

}
```

Las llamadas `wait` y `exit`

`<unistd.h>`

- La función `wait()` forzará a un proceso padre que espere a que un proceso hijo se detenga o termine
- La función regresa el PID del hijo o `-1` en caso de error. El estado de la salida del hijo es regresado en status

```
int wait() (int *status)
pid_t waitpid (pid_t pid, int *status-ptr,
int options)
```

- La función `exit()` termina el proceso que llama a esta función y regresa en la salida el valor de status. Tanto UNIX como los programas bifurcados de C, pueden leer el valor de status

```
void exit(int status)
void abort(void)
```

Resumiendo

- La llamada del sistema `fork()` duplica le proceso padre, asignándole un PID único al hijo
- La diferentes variantes de `exec()` permiten ejecutar programas externos
- Las llamdas `exit()` y `abort()` permiten terminar la ejecución de un programa
- Las llamadas `wait()`y `waitpid()` permiten a un proceso padre, esperar a que un subproceso termine
- Las llamdas `getpid()` y `getppid()` regresan el PID del proceso y el PID del proceso padre, respectivamente