Informática

Unidad 3: Interacción con el hardware y el Sistema Operativo

Ingeniería en Mecatrónica

Facultad de Ingeniería
Universidad Nacional de Cuyo

Dr. Ing. Martín G. Marchetta
mmarchetta@fing.uncu.edu.ar

UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERIA
en acción continua...

3B – Hilos y Procesos

3B – Hilos y Procesos

- La programación con multithreading y/o multiprocessing permite ejecutar instrucciones en pseudo paralelismo

- Esto es especialmente útil cuando el software realiza mucha entrada/salida, además del procesamiento de datos en la CPU

- Mientras un thread o proceso está esperando que se complete una operación de entrada/salida, otros pueden utilizar la CPU, logrando un mayor ***throughput*** (cantidad de procesamiento promedio realizado durante un cierto período de tiempo)

- Usos en aplicaciones mecatrónicas
  - **Plataformas de simulación**: Generalmente se requirere una operación en pseudo-paralelismo de la plataforma que simula el hardware, y del software que opearía dicho hardware
  - **Sistemas de control**: los sistemas de control requieren, al menos, interacción con actuadores (además, sensores si es un control con lazo cerrado). La interacción con sensores, actuadores y el procesamiento del algoritmo de control pueden paralelizarse, ganando en performance
  - **Procesamiento distribuido/paralelo**: cuando los algoritmos son paralelizables, es posible utilizar más de una unidad de cómputo para ejecutar en paralelismo real (ya sea en múltiples CPU, o bien en múltiples computadoras en un esquema Cluster o Grid)
  - **Sistemas híbridos deliberativos/reactivos**: Los sistemas inteligentes complejos pueden involucrar componentes deliberativos (ej: planificación con un horizonte de tiempo mediano/largo), y componentes reactivos (ej: sistema de control) o bien componentes de control “de bajo nivel” (ej: mover un brazo de una posición **X** a una posición **Y**). Estos componentes se pueden paralelizar utilizando multithreading/multiprocessing

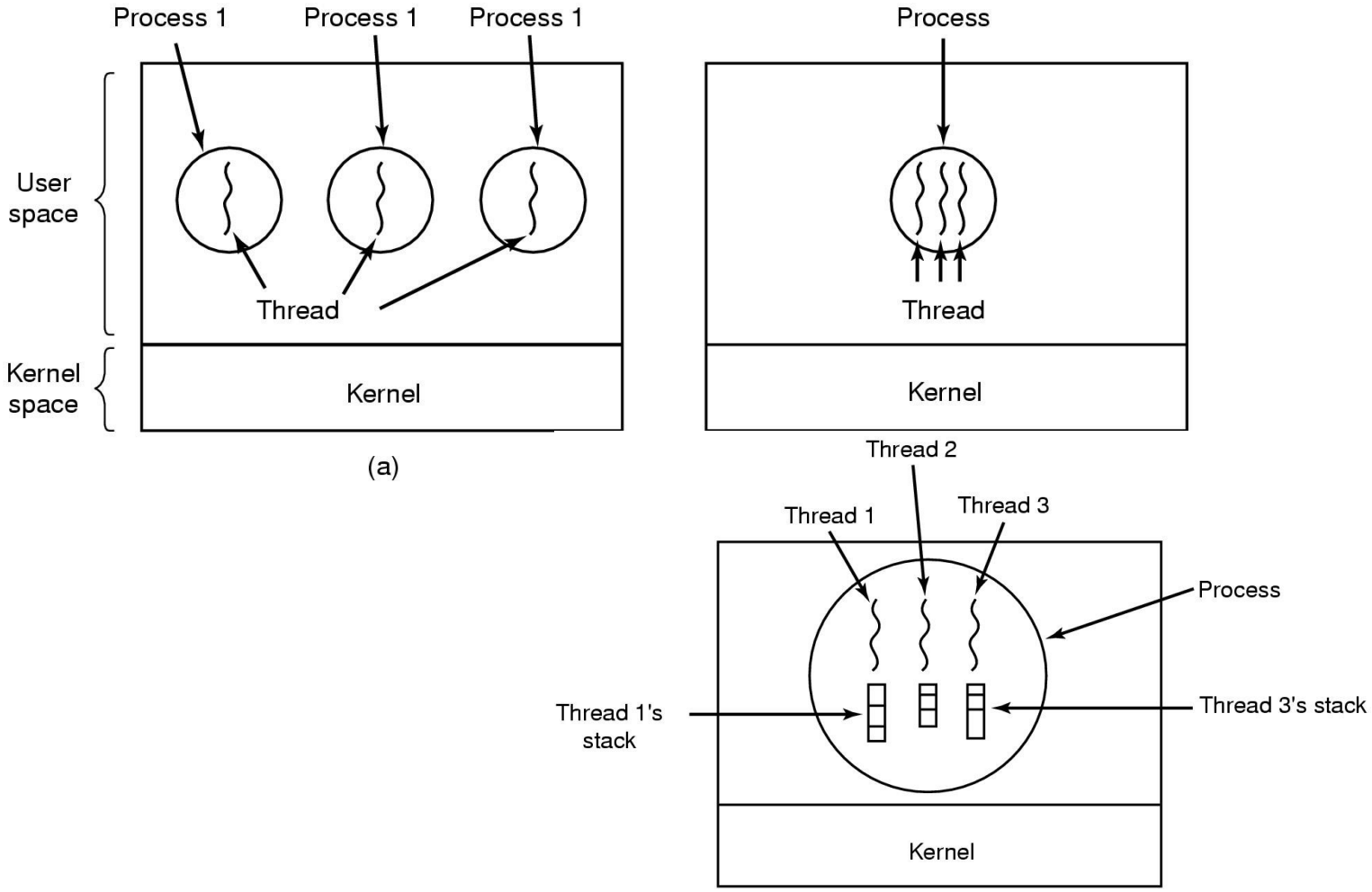
3B – Hilos y Procesos (repaso)

- Hilos (threads): También llamados ***procesos ligeros***
  - Los threads son *hilos de ejecución* paralelos
  - Cada proceso tiene un espacio de direcciones independiente del de los otros procesos
  - En cambio, todos los hilos de un mismo proceso
    - Comparten las mismas instrucciones
    - Comparten el mismo espacio de memoria (variables)
    - Pero cada uno tiene un valor independiente para el **PC**, **registros**, **pila** y **estado**, con lo cual la ejecución de cada uno puede variar

- Los threads múltiples comparten algunas de las características de los procesos múltiples:
  - Pueden estar en los mismos estados básicos (en ejecución, bloqueado, listo)
  - Permiten aprovechar mejor la CPU cuando un hilo está bloqueado esperando por datos del disco o la red, u otro hilo o proceso

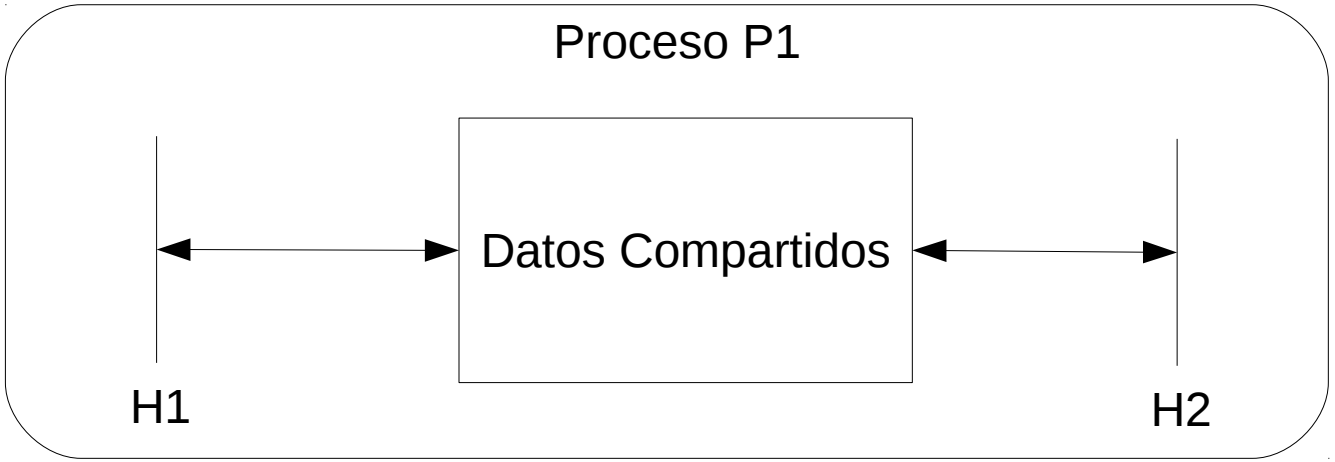
3B – Hilos y Procesos (repaso)

- Threads: relación con los procesos



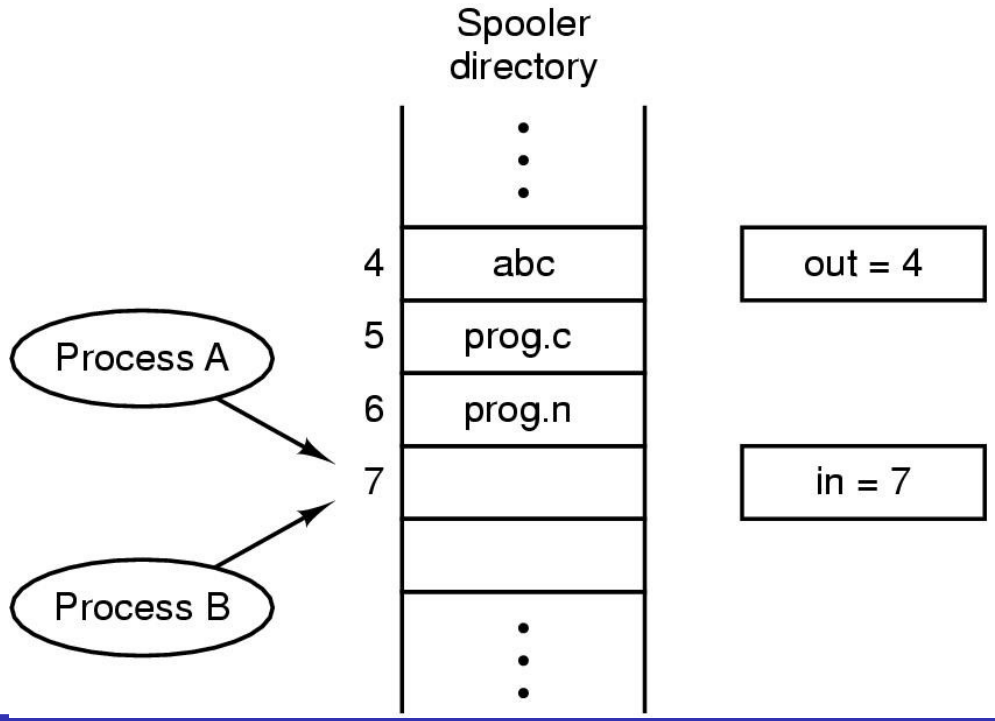
3B – Hilos y Procesos (repaso)

- El hecho de que los threads comparten el espacio de memoria hace más simple el intercambio de datos entre los hilos



- En cambio, la comunicación entre procesos independientes generalmente requiere el uso de algún mecanismo explícito. Ej: pipes, sockets
- En ambos casos, la sincronización es muy importante para evitar problemas de concurrencia

- Condiciones de competencia
  - Los hilos/procesos pueden tener problemas cuando acceden a recursos compartidos
  - Estas situaciones se llaman ***condiciones de competencia***



3B – Hilos y Procesos (repaso)

- Sincronización
  - Existen partes del código que no presentan riesgo ante la concurrencia, mientras que otras áreas del código pueden potencialmente generar problemas
  - Las áreas del código problemáticas son las que leen/modifican estructuras de datos compartidas o acceden a recursos compartidos: estas son las llamadas ***secciones críticas***
  - Nunca 2 procesos o threads deben estar en la misma sección crítica al mismo tiempo, puesto que los resultados son impredecibles
- Un mecanismo de sincronización típico es el uso de ***semáforos***
- Los semáforos son variables especiales que pueden ser modificados con funciones especiales mediante **operaciones atómicas**
- La idea es que antes de entrar a una sección crítica, el thread o proceso debe tratar de “tomar” el semáforo; si el semáforo está libre, el thread/proceso puede entrar en su sección crítica; si no, se bloquea esperando que el proceso que tiene el semáforo lo libere

- Dependencias
  - Biblioteca pthread:
    - Se debe incluir pthread.h
    - Se debe compilar con el parámetro -lpthread (parámetro del linker)
      - Esto puede hacerse en la línea de comandos (si se está compilando manualmente utilizando gcc)

```
gcc -lpthread programa.c -o programa
```

      - Configurando la biblioteca pthread en el IDE. Por ejemplo, en Eclipse, esto se hace de la siguiente manera:
        - Ir a las propiedades del proyecto
        - C/C++ Build
        - Settings
        - GCC Linker Libraries
        - Presionar el botón “+” (arriba a la derecha)
        - Agregar la biblioteca “pthread”

- Estructuras de datos
  - ID de un thread: Cada vez que se crea un thread se le asigna un ID único, que se debe utilizar para realizar operaciones subsiguientes con el thread

    ```
    pthread_t
    ```

  - ID de un mutex: Para sincronizar threads se usa un tipo especial de semáforo llamado mutex

    ```
    pthread_mutex_t
    ```

- Funciones
  - Creación de un thread

    ```
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                       void *(*start_routine)(void*), void *arg);
    ```

  - Re-unificación de un thread cuando termina su ejecución

    ```
    int pthread_join(pthread_t thread, void **value_ptr);
    ```

- Ejemplo:

```
void *funcion(void *arg) {
    printf("Esto es un thread!");
}
int main(){
    pthread_t t1;
    pthread_create(&t1, NULL, funcion, NULL);
    ...
    pthread_join(t1, NULL); //re-unificamos el thread una vez que
    termina
    return 0;
}
```

Cuando se crea el thread, se llama a esta función

- Mutex (tipo especial de semáforo): Inicialización
  - Inicialización estática

```
pthread_mutex_t mutext = PTHREAD_MUTEX_INITIALIZER;
```

  - Inicialización dinámica

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

  - Destrucción de un mutex (esto no libera la memoria!)

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Uso de mutex
  - Bloqueo de un mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - Liberación del bloqueo sobre un mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Ejemplo

```
int contador=0;

pthread_mutex_t contador_mutex=PTHREAD_MUTEX_INITIALIZER;

void *funcion(void *arg) {
    int i;
    int temp;
    for(i = 0; i < 10; i++){
        pthread_mutex_lock(&contador_mutex);
        printf("(Seccion critica) Contador: %d", contador++);
        pthread_mutex_unlock(&contador_mutex);
    }
}
...
```

- Sincronización, una herramienta más: Variables de condición
  - Son variables especiales que un thread puede usar para requerir una ***notificación*** de otro thread ante un evento
  - Para ello,
    1. Un thread T1 bloquea el mutex
    2. Luego T1 verifica una ***condición***; si la condición no se da, T1 queda bloqueado ***en espera sobre la condición***, lo cual **libera** el mutex para que otro thread lo pueda utilizar mientras T1 espera a que se de la condición
    3. Luego, cuando otro thread T2 satisface la condición que T1 espera, se lo notifica mediante la variable de condición (le envía una ***señal***)
    4. Finalmente, T1 es “despertado”, y el mecanismo de variable de condición intenta bloquear el mutex automáticamente; cuando lo logra, T1 continúa con su ejecución

- Variables de condición: implementación
  - Declaración / inicialización de una variable de condición

```
pthread_cond_t condicion=PTHREAD_COND_INITIALIZER;
```

  - Verificación/bloqueo en espera de la condición

```
pthread_cond_wait(&condicion, &mutex);
```

Notese que la función recibe como parámetros tanto la variable de condición sobre la que el thread quedará bloqueado, como el mutex que debe ser liberado

- Variables de condición: implementación
  - Cuando un thread T2 satisface la condición que T1 espera, se lo notifica mediante la variable de condición. Hay 2 opciones
    - Notificación a uno de los threads

```
//Notifica a 1 de los threads que esperan
pthread_cond_signal(&condicion);
```

    - Notificación a todos los threads

```
//Notifica a todos los threads que esperan
pthread_cond_broadcast(&condicion);
```

3B – Hilos y Procesos

- Multiprocessing
  - A diferencia de los threads, cuando un proceso crea un proceso hijo
    - este hijo **no** comparte el mismo espacio de direcciones y la misma imagen del núcleo
    - el hijo es una **copia exacta** del padre al momento de crearse
  - El proceso hijo contiene las mismas variables y estado del padre al momento de crearse, y luego los 2 procesos evolucionan independientemente
  - Desventaja:
    - requiere más procesamiento “administrativo” (los threads son “procesos ligeros”)
  - Ventajas:
    - Robustez: si un thread falla, todo el proceso falla (no se puede “matar” un thread independientemente del proceso); si un proceso hijo falla, el padre y cualquier otro “hermano” del proceso pueden seguir ejecutándose
    - Los procesos independientes pueden correr en unidades de cómputo diferentes
    - Los procesos independientes reciben su ***time slice*** directamente del SO; los threads reciben su ***time slice*** del proceso, el cual recibe sólo 1 ***time slice*** del SO
    - Multiprocessing se soporta a nivel de SO, mientras que multithreading, ***en general***, se soporta a través de bibliotecas en espacio de usuario

- Multiprocessing: implementación
  - Dependencias: también requieren linkear con la biblioteca pthread, al igual que los threads
  - Estructuras de datos
    - ID de un proceso (pid, process id): Cada vez que se crea un proceso hijo, se le asigna un ID único (lo hace el SO), que se debe utilizar para realizar operaciones subsiguientes con el proceso

      ```
      pid_t
      ```

  - Funciones
    - Creación de un proceso hijo

      ```
      pid_t fork(void);
      ```

- Multiprocessing: implementación
  - La función `fork()` devuelve
    - 0 para el proceso hijo
    - El **pid** del nuevo proceso hijo, para el padre
  - Dado que los dos procesos son idénticos en ese punto, luego de crear el hijo, cada uno debe verificar si es el hijo o el padre. Ej:

```
pid_t pid_hijo;
pid_hijo = fork();

if(pid_hijo == -1){
   perror("Error al crear proceso hijo");
   exit 1;
}
else if(pid_hijo == 0){
   //Codigo que ejecutara solamente el hijo
   …
}
else{
   //Codigo que ejecutara solamente el padre
   ...
}
```

- Multiprocessing: implementación
  - El proceso padre puede bloquearse explícitamente hasta que el proceso hijo termina mediante la función `waitpid()`

```
pid_t pid_hijo;
pid_hijo = fork();
…
else if(pid_hijo != 0){
    //Codigo del padre
    ...
    waitpid(pid_hijo); //Esperamos a que el hijo termine
    ...
}
```

- Multiprocessing: implementación
  - Luego de `fork()`, los procesos padre e hijo tienen las mismas variables, pero los cambios que uno realiza no son visibles para el otro
  - En la mayoría de los casos, se requiere un mecanismo para que los procesos puedan intercambiar información: Comunicación Inter-Procesos (Inter-Process Communication, IPC).
  - Existen varios mecanismos para esto. Los más comunes que siguen el estándar POSIX son ***signals*** (“señales”, tambien llamadas “interripciones de software”) ***pipes*** (tuberías) y ***sockets*** (“enchufes”, como metáfora para referirse a puntos de conección sobre un protocolo de red)

- Multiprocessing: implementación
  - Sincronización mediante semáforos POSIX
    - Declaración de un semáforo

```
sem_t* semaforo;
```

    - Creación/apertura de un semáforo con nombre

```
#define ID_SEMAFORO "/semaforo_prueba"

semaforo = sem_open(ID_SEMAFORO, O_CREAT, S_IRUSR |
S_IWUSR | S_IRGRP | S_IWGRP, 1);
```

Parámetros
ID_SEMAFORO: nombre del semaforo a crear
O_CREAT: indica que se cree el semáforo si no existía (si no, se abre)
Permisos: formato rwx para usuario, grupo y el resto, o bien constantes
Valor inicial del semáforo

- Multiprocessing: implementación
  - Sincronización mediante semáforos POSIX
    - Bloqueo de un semáforo (down sobre el semáforo)

      ```
      int sem_wait(sem_t *sem);
      ```

    - Desbloqueo de un semáforo (up sobre el semáforo)

      ```
      int sem_post(sem_t *sem);
      ```

    - Cierre del semáforo

      ```
      int sem_close(sem_t *sem);
      ```

    - Liberación de los recursos del semáforo

      ```
      int sem_unlink(const char *name);
      ```

      NOTA: `sem_unlink` recibe el nombre, no el apuntador