



**Facultad de Ingeniería en
Electricidad y Computación**

Programación de Sistemas

CCPG1051

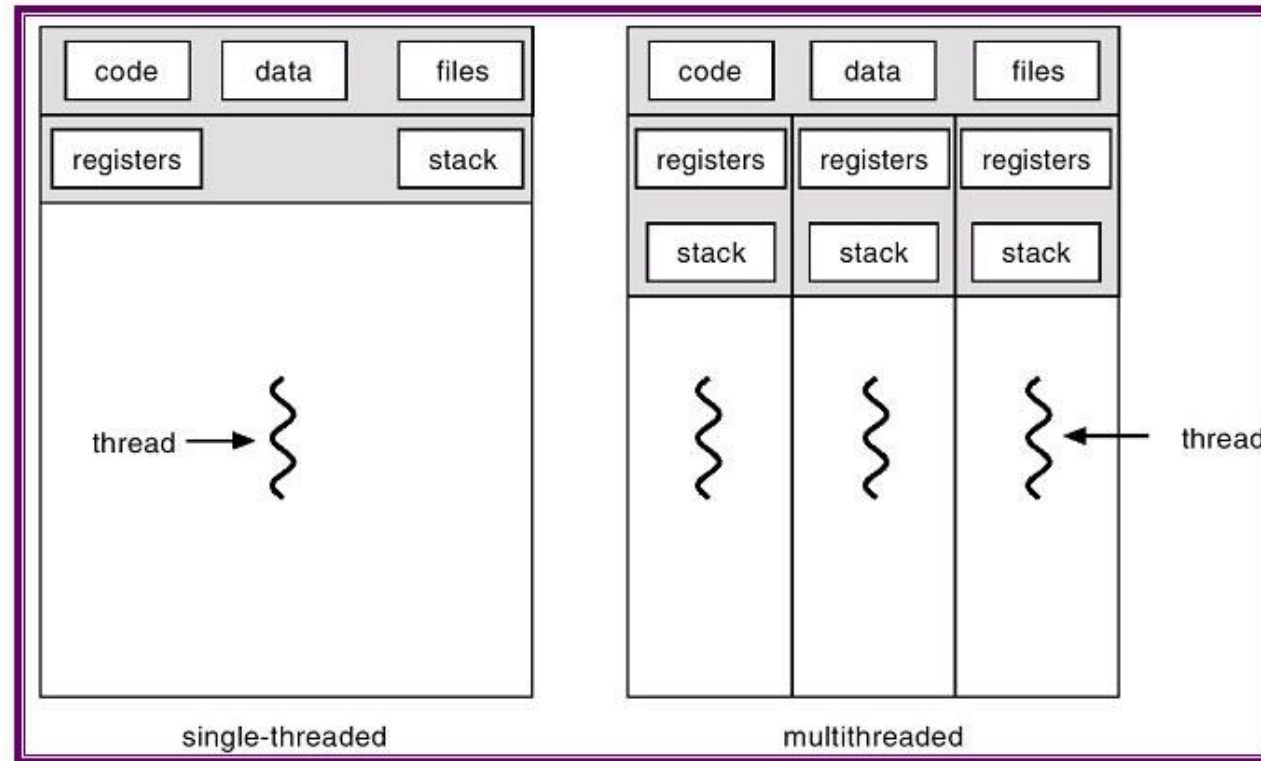
Federico Domínguez, PhD.

Unidad 6 – Sesión 4: Consideraciones con hilos

Agenda

- Variables compartidas
- Secciones críticas
- Semáforos y POSIX
- Thread safety
- Tipos de funciones *thread-unsafe*
- Funciones *reentrants*
- Races

Un proceso puede mantener más de un flujo lógico de manera concurrente mediante el concepto de **hilos**.



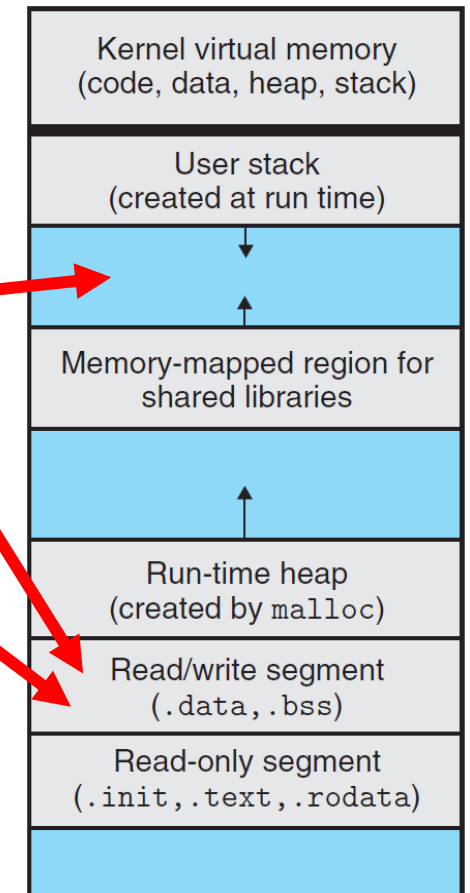
Un hilo comparte el espacio de memoria virtual del proceso al cual pertenece.

La ventaja del modelo de concurrencia con hilos es la capacidad de **compartir variables** entre hilos.

Las variables en C se pueden dividir de acuerdo a como son almacenadas en la memoria virtual:

- **Globales:** Declaradas fuera de una función. Almacenadas en *.data* o *.bss* de la memoria virtual y pueden ser referenciadas por cualquier hilo. **Una instancia por proceso.**
- **Locales automáticas:** Declaradas dentro de una función sin `static`. Almacenadas en el *stack* de cada hilo, cada hilo tiene su propia copia. **Una instancia por hilo.**
- **Locales estáticas:** Declaradas dentro de una función con `static`. Almacenadas en *.data* o *.bss* de la memoria virtual y pueden ser referenciadas por cualquier hilo. **Una instancia por proceso.**

Una variable es **compartida** si y solo si una instancia de la misma es compartida por más de un hilo.



```
1  #include "csapp.h"
2  #define N 2
3  void *thread(void *vargp);
4
5  char **ptr;  /* Global variable */
6
7  int main()
8  {
9      int i;
10     pthread_t tid;
11     char *msgs[N] = {
12         "Hello from foo",
13         "Hello from bar"
14     };
15
16     ptr = msgs;
17     for (i = 0; i < N; i++)
18         Pthread_create(&tid, NULL, thread, (void *)i);
19     Pthread_exit(NULL);
20 }
21
22 void *thread(void *vargp)
23 {
24     int myid = (int)vargp;
25     static int cnt = 0;
26     printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
27     return NULL;
28 }
```

Compartir variables entre hilos introduce complicaciones inherentes a la programación con concurrencia.

```
code/conc/badcnt.c
1  #include "csapp.h"
2
3  void *thread(void *vargp); /* Thread routine prototype */
4
5  /* Global shared variable */
6  volatile int cnt = 0; /* Counter */
7
8  int main(int argc, char **argv)
9  {
10     int niters;
11     pthread_t tid1, tid2;
12
13     /* Check input argument */
14     if (argc != 2) {
15         printf("usage: %s <niters>\n", argv[0]);
16         exit(0);
17     }
18     niters = atoi(argv[1]);
19
20     /* Create threads and wait for them to finish */
21     Pthread_create(&tid1, NULL, thread, &niters);
22     Pthread_create(&tid2, NULL, thread, &niters);
23     Pthread_join(tid1, NULL);
24     Pthread_join(tid2, NULL);
25
26     /* Check result */
27     if (cnt != (2 * niters))
28         printf("BOOM! cnt=%d\n", cnt);
29     else
30         printf("OK cnt=%d\n", cnt);
31     exit(0);
32 }
33
34 /* Thread routine */
35 void *thread(void *vargp)
36 {
37     int i, niters = *((int *)vargp);
38
39     for (i = 0; i < niters; i++)
40         cnt++;
41
42     return NULL;
43 }
```

code/conc/badcnt.c

Demo *badcnt.c*

Al inspeccionar el código que accede e incrementa la variable compartida en ensamblador, se puede apreciar que para lograr esto se necesitan como mínimo tres instrucciones.

C code for thread i

```
for (i=0; i < niters; i++)  
    cnt++;
```

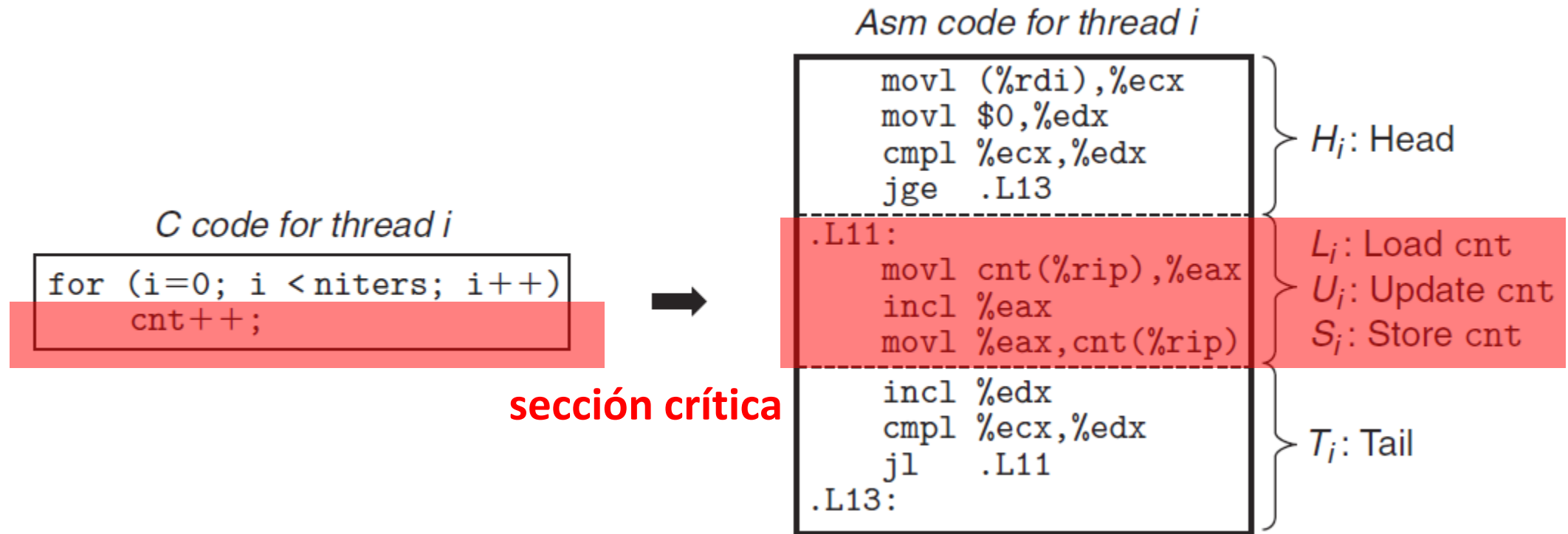


Asm code for thread i

<pre>movl (%rdi),%ecx movl \$0,%edx cmpl %ecx,%edx jge .L13</pre>	}	H_i : Head
<pre>.L11: movl cnt(%rip),%eax incl %eax movl %eax,cnt(%rip)</pre>		
<pre>incl %edx cmpl %ecx,%edx jl .L11 .L13:</pre>	}	T_i : Tail

L_i : Load cnt
 U_i : Update cnt
 S_i : Store cnt

Al inspeccionar el código que accede e incrementa la variable compartida en ensamblador, se puede apreciar que para lograr esto se necesitan como mínimo tres instrucciones.



El orden en que ambos hilos ejecutan las instrucciones que acceden y modifican una variable compartida es crítico para obtener resultados correctos.

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

(a) Correct ordering

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

(b) Incorrect ordering

El orden en que ambos hilos ejecutan las instrucciones que acceden y modifican una variable compartida es crítico para obtener resultados correctos.

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

(a) Correct ordering

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

(b) Incorrect ordering

El concepto del semáforo ofrece una solución a la sincronización de acceso a secciones críticas de código.

Un semáforo, s , es una variable global no-negativa la cual puede ser manipulada por tan solo dos operaciones: P y V .

- $P(s)$: Si s es diferente de cero, P decrementa s y retorna inmediatamente. Si s es cero, el hilo se bloquea hasta que una llamada a V lo desbloquee. En ese caso, el hilo decrementa el valor de s y P retorna.
- $V(s)$: V incrementa s en 1. Si existen hilos esperando en P , V reinicia exactamente un solo hilo, el cuál procede a decrementar s y continua.

Las operaciones P y V son indivisibles (atómicas) y esto es usualmente garantizado por hardware o el sistema operativo.

La operación V no define que hilo bloqueado en P debe reactivar, esto no es necesariamente predecible.

El estándar POSIX define varias funciones para la manipulación de semáforos.

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, 0, unsigned int value);
```

```
int sem_wait(sem_t *s);    /* P(s) */
```

```
int sem_post(sem_t *s);    /* V(s) */
```

Returns: 0 if OK, -1 on error

Las funciones “wrapper” usadas en el libro guía mantienen la nomenclatura original.

```
#include "csapp.h"
```

```
void P(sem_t *s); /* Wrapper function for sem_wait */
```

```
void V(sem_t *s); /* Wrapper function for sem_post */
```

Returns: nothing

Los semáforos proveen una manera conveniente para asegurar acceso exclusivo a variables compartidas.

Si inicializamos s en 1, s se convierte en un semáforo binario.

Si rodeamos una sección crítica con $P(s)$ y $V(s)$ de un semáforo binario, tenemos una zona de exclusión mutua.

El uso de un semáforo de esta forma, se conoce como *mutex* (mutual exclusión).

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}
```

Demo *goodcnt.c*

Seguridad en hilos (Thread Safety)

Una función es *thread-safe* si y solo si produce resultados correctos cuando es llamada de manera concurrente por diferentes hilos.

Al desarrollar una aplicación con concurrencia debemos asegurarnos que todas las funciones sean *thread-safe*.

Existen varias formas como una función podría ser *thread-unsafe*...

Tipos de funciones *thread-unsafe*

Tipo 1: Funciones que no protegen variables compartidas con semáforos o métodos equivalentes.

Tipo 2: Funciones que mantienen estados a través de múltiples invocaciones. Usan variables locales estáticas o similares.

Tipo 3: Funciones que retornan punteros a variables estáticas.

Tipo 4: Funciones que llaman a otras funciones *thread-unsafe*.

Thread-unsafe tipo 2

code/conc/rand.c

```
1  unsigned int next = 1;
2
3  /* rand - return pseudo-random integer on 0..32767 */
4  int rand(void)
5  {
6      next = next*1103515245 + 12345;
7      return (unsigned int)(next/65536) % 32768;
8  }
9
10 /* srand - set seed for rand() */
11 void srand(unsigned int seed)
12 {
13     next = seed;
14 }
```

code/conc/rand.c

Gestión de una función *thread-unsafe* tipo 3.

code/conc/ctime_ts.c

```
1  char *ctime_ts(const time_t *timep, char *privatep)
2  {
3      char *sharedp;
4
5      P(&mutex);
6      sharedp = ctime(timep);
7      strcpy(privatep, sharedp); /* Copy string from shared to private */
8      V(&mutex);
9      return privatep;
10 }
```

code/conc/ctime_ts.c

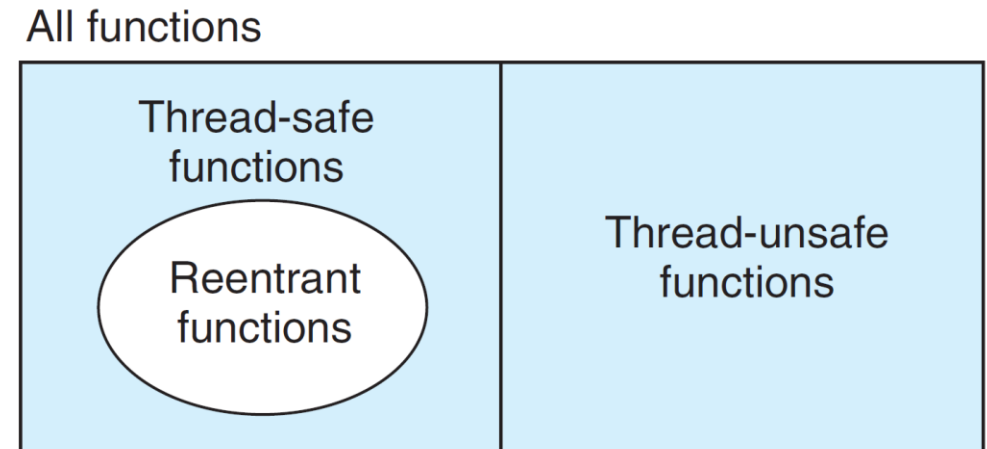
Demo función thread unsafe

Funciones re-entrantes

Una función es re-entrante (*reentrant*) cuando no referencia a ningún tipo de variables o recursos compartidos.

El nombre proviene de la propiedad que una función re-entrante puede ser interrumpida en cualquier momento durante su ejecución y ser ejecutada por otro hilo (re-entrada) de manera segura.

No todas las funciones *thread-safe* son re-entrantes...



Funciones re-entrantes

Una función es re-entrante explícita si no usa ninguna variable compartida y no usa argumentos pasados por referencia (usando punteros).

Una función es re-entrante implícita si no usa ninguna variable compartida pero contiene parámetros pasados por referencia.

La única forma de corregir una función *thread-unsafe* tipo 2 es reescribirla como re-entrante.

```
1  /* rand_r - a reentrant pseudo-random integer on 0..32767 */
2  int rand_r(unsigned int *nextp)
3  {
4      *nextp = *nextp * 1103515245 + 12345;
5      return (unsigned int)(*nextp / 65536) % 32768;
6  }
```

code/conc/rand_r.c

¿Es esta función re-entrante?

code/conc/ctime_ts.c

```
1  char *ctime_ts(const time_t *timep, char *privatep)
2  {
3      char *sharedp;
4
5      P(&mutex);
6      sharedp = ctime(timep);
7      strcpy(privatep, sharedp); /* Copy string from shared to private */
8      V(&mutex);
9      return privatep;
10 }
```

code/conc/ctime_ts.c

Algunas funciones de librerías estándares en sistemas UNIX son *thread-unsafe*, sin embargo se proveen versiones *thread-safe* y son demarcadas con el sufijo `_r`.

Thread-unsafe function	Thread-unsafe class	Unix thread-safe version
<code>rand</code>	2	<code>rand_r</code>
<code>strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>

Figure 12.39 Common thread-unsafe library functions.

Demo Race

Races

Una carrera (*race*) es una de las causas de resultados erróneos más comunes en sistemas concurrentes.

Una carrera ocurre cuando el resultado correcto de un programa depende de que hilo llega primero a un punto específico del flujo lógico del programa.

Referencias

Libro guía Computer Systems: A programmer's perspective. Secciones 12.4,5,7