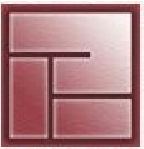


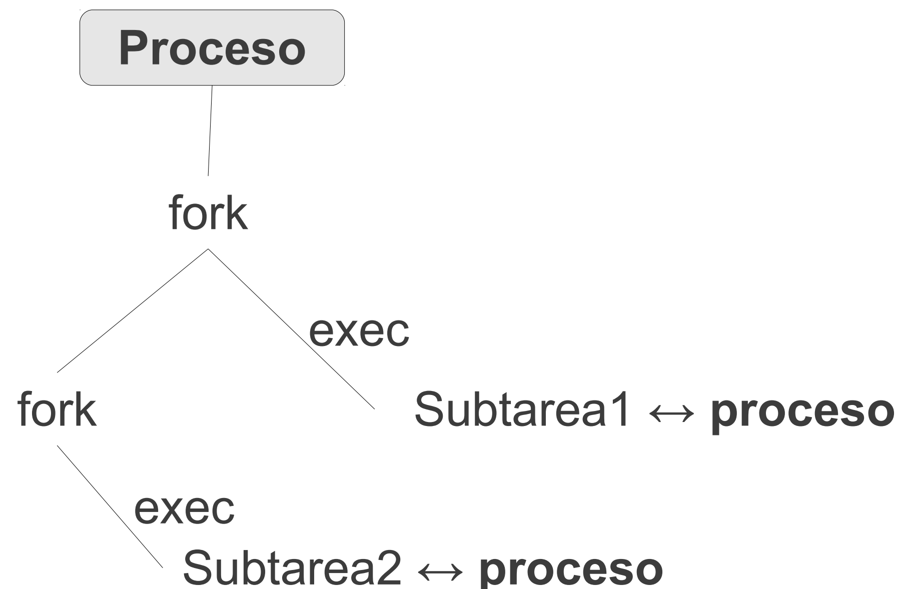
Instituto
Politécnico
Superior

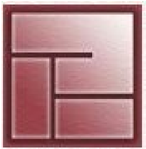
THREADS

Instituto Politécnico Superior

Introducción

- Los procesadores actuales permiten realizar muchas tareas en simultáneo
- En los sistemas la ejecución de las tareas puede intercalarse en un sólo procesador, y en casos de multi-procesadores, ejecutarse en paralelo
- Cuando dividimos un programa en múltiples subtareas las “enviamos” al procesador en forma de procesos
- Diseño de programa





Introducción

- Una manera de ver a un proceso es como una agrupación de recursos (archivos abiertos, procesos hijos, handlers de señales, etc.), tal agrupación es más fácil de administrar al tratarla como un único elemento

- El otro concepto asociado a los procesos es el hilo de control o *thread*.

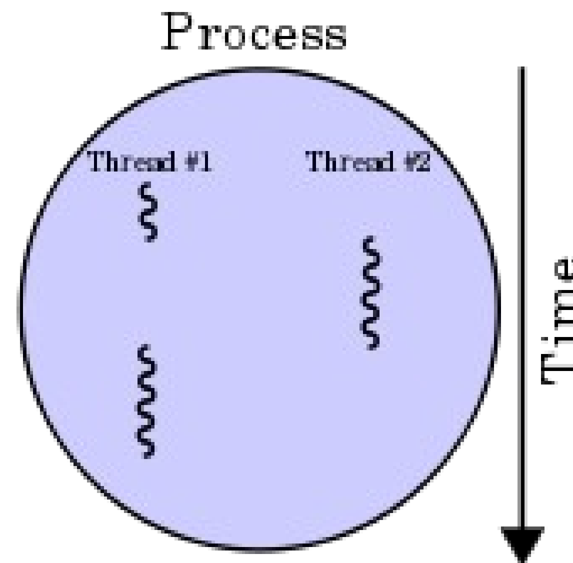
- Un thread tiene
  - un contador de programa (PC), que señala la próxima instrucción a ejecutar.
  - registros, que contienen variables de uso actual.
  - un stack, con un marco por cada proceso invocado que no ha retornado aun.

- Un thread se ejecuta en el contexto de un proceso, pero son conceptos diferentes.
  - Un proceso es una agrupación de recursos
  - Un thread es lo que se planifica y al cual se brinda ciclos de CPU

Instituto
Politécnico
Superior

Introducción

- Cada proceso cuenta con un espacio de direcciones y uno o, posiblemente, más hilos (threads) de control.
  Estos hilos pueden verse como si fueran procesos independientes (excepto por el espacio de direcciones compartido)



- Dado que los threads comparten muchas de las propiedades de los procesos son llamados también *lightweight process*

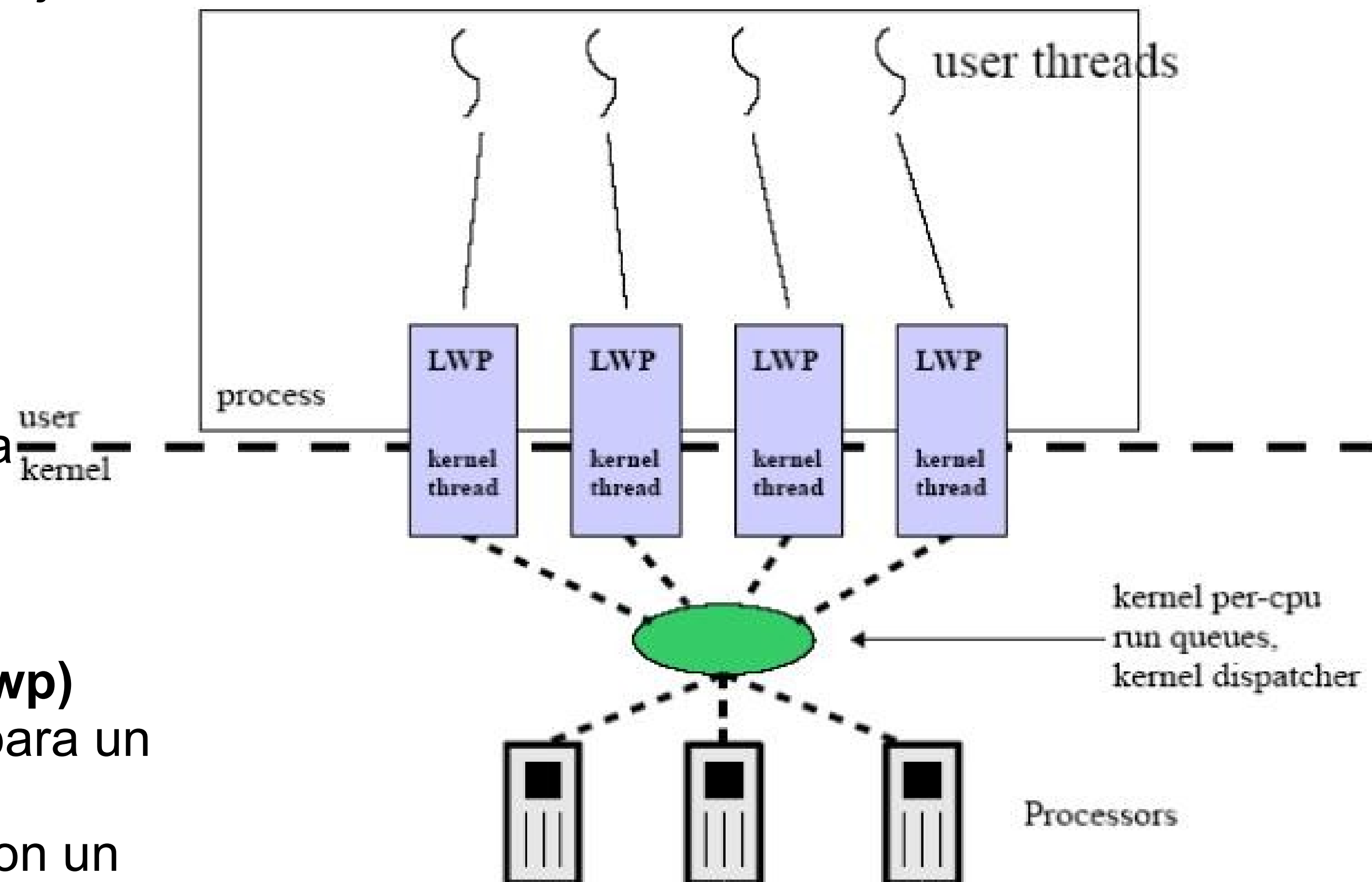
- El término *multithreading* es empleado para describir la situación donde pueden existir dos o más threads en un mismo proceso

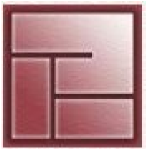
Instituto Politécnico Superior

Threads (LWPs - Solaris 10)

Kernel thread model

- **kernel threads**
  Es lo que es planificado/ejecutado en un procesador
- **user threads**
  Hilo de ejecución de un proceso de usuario
- **process**
  El objeto que representa el entorno de ejecución de un programa
- **lightweight process (lwp)**
  Contexto de ejecución para un thread de usuario.
  Asocia un user thread con un kernel thread





Threads

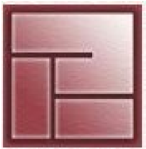
- Un proceso posee un espacio de direcciones, el cual es compartido por sus threads, por tanto cada thread puede leer o escribir en las direcciones del proceso, hasta puede limpiar el stack de otro

- No hay protección entre los threads...
  ... pues no es necesario, a diferencia de un número de procesos competitivos, el proceso que contiene los threads pertenece a un único usuario, el cual adoptó el esquema de threads para que los mismos cooperen

- Además de compartir el mismo espacio de direcciones, los threads comparten los archivos abiertos, los procesos hijos, señales, etc.

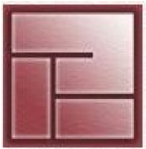
Elementos por proceso
Espacio de direcciones
Variables Globales
Archivos Abiertos
Procesos hijos
Señales y handlers de señales
Información de accounting

Elementos por threads
Contador de programa
Registros
Stack
Estado



Threads

- Como los procesos los threads pueden estar en un único estado:
  - en ejecución
  - bloqueado o en espera
  - listo
  - terminado
- Cada thread tiene su propia copia de los registros
- Los threads son planificados para su ejecución, ocurren cambios de contexto, los cuales son mucho más “económicos” que entre procesos
- La creación y gestión de threads es más eficiente que para los procesos

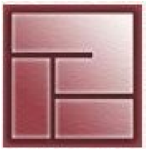


Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Creación de 50,000 process/thread

Valores obtenidos con time

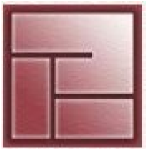
Resultados expresados en segundos



Ejemplo

```
#includes....
void hacer_una_cosa() {
    int i, j, k;
    //codigo
}

void hacer_otra_cosa() {
    int u, v, w;
    //codigo
}

void resumen(int unas, int otras) {
    int total = unas + otras;
    imprimir(total);
}

int r1=0, r2=0;
```

```
int main() {
    hacer_una_cosa(&r1);
    hacer_otra_cosa(&r2);
    resumen(r1, r2);

    return 0;
}
```



Layout del proceso en Memoria

Recursos del sistema para administrar el proceso

ESPACIO DE MEMORIA VIRTUAL

Registros SP
PC

THREAD 2

```
hacer_otra_cosa()
u stack
v
w
```

Registros SP
PC

THREAD 1

```
hacer_una_cosa()
i stack
j
k
```

Identidad
PID,PPID
UID
GID

```
main()
  hacer_una_cosa
  hacer_otra_cosa
  return 0
}
```

Segmento de Texto
Area read only

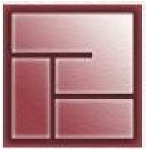
Recursos
File descriptors
Sockets
Señales

```
r1
r2
```

Area r/w para datos globales
Segmento de Datos

Segmento de Sistema

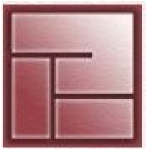
Heap (malloc/free)



Organización de threads. Modelos

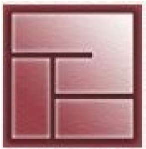
• *Boss/Worker Model*

- Un thread oficia de "jefe" asignando tareas a threads trabajadoras
- Cada thread trabajadora realiza tareas hasta finalizar, interrumpiendo al jefe para indicar que se encuentra disponible nuevamente.
- Alternativamente, el jefe consulta periódicamente a los threads para ver si se encuentran ociosas



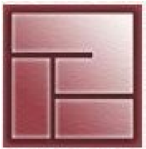
Organización de threads. Modelos

• *Peer Model*
- También llamado work crew model
- Múltiples threads trabajan conjuntamente para realizar una tareas única
- La tarea es divided en piezas que se ejecutan en paralelo, cada thread ejecuta una pieza
- Ejemplo: thread=personas tarea=limpiar una casa



Organización de threads. Modelos

• *Pipeline*
- Una tarea es dividida en pasos
- Los pasos deben ser realizados en secuencia para producir un único resultado
- El trabajo realizado en cada paso (excepto para el primero y el último) está basado en el resultado generado por el paso anterior y es necesario para el siguiente
- Un ejemplo es cualquier proceso de ensamblado, como ser la línea de producción de automóviles.



Pthreads

- POSIX Threads

- Provee rutinas para
  - Creación y destrucción de threads
  - Sincronización
  - Scheduling, gestión de prioridades
  - Administración

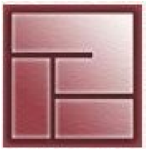
- Funciones definidas en **`/usr/include/pthread.h`**

- En el ejemplo — Ejecuciones válidas

En el ejemplo	Ejecuciones válidas	
`hacer_una_cosa`	1ro	2do
`hacer_otra_cosa`	2do	1ro
`main`	3ro	3ro

- Se dice que un programa posee la propiedad de **Paralelismo Potencial** si se compone de sentencias que pueden ejecutarse en cualquier orden sin afectar el resutaldo final

Paralelismo Potencial $\Rightarrow$ Ejecución más eficiente



Pthreads: Funciones

```
int pthread_create (pthread_t *thr, const pthread_attr_t *attr,
                    void* (*rutina) (void*), void *arg)
```

thr
puntero en el cual la función retorna el identificador del nuevo thread, es un *unsigned long int*

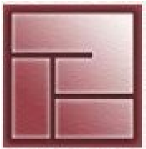
attr
puntero a una *union* que define atributos del nuevo thread. NULL: acepta atributos por defecto

rutina
Puntero a la rutina donde comenzará la ejecución del nuevo thread

arg
Puntero a un parámetro a ser pasado a la rutina de inicio

Valor de retorno
0: OK
≠ 0: identificador de error

Nota: Los tipos de la librería están definidos en `/usr/include/bits/pthreadtypes.h`

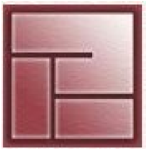


Pthreads: Funciones

La función **`pthread_exit`** finaliza el thread invocante

```
void pthread_exit (void *value)
```

`value`
Valor de retorno para todo thread que realiza un **`pthread_join`** sobre este



Pthreads: Funciones

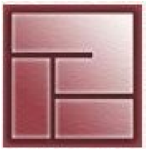
Los procesos poseen relaciones padre-hijo y mediante wait/waitpid el proceso padre espera la finalización del hijo

Con la función **`pthread_join`** podemos hacer que un thread espere por la finalización de otro

```
int pthread_join (pthread_t thr, void **value_ptr)
```

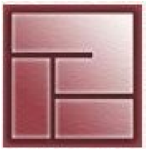
`thr`
Thread por el cual esperar

`value_ptr`
Recibe el valor de retorno del thread esperado



Pthreads: Ejemplo

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* mensaje1(void *x) {
    printf("HOLA ");
}

void* mensaje2(void *x) {
    printf("MUNDO\n");
}

int main() {
    pthread_t p1, p2;
    pthread_create(&p1, 0, mensaje1, NULL);
    pthread_create(&p2, 0, mensaje2, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    return 0;
}
```



Pthreads: Race Condition

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
int asistentes = 0;

void* puerta1(void *x) {
    int i, k;
    for(k=0; k<1000; k++) {
        i = asistentes;
        i = i + 1;
        asistentes = i;
    }
    return NULL;
}

void* puerta2(void *x) {
    int j, k;
    for(k=0; k<1000; k++) {
        j = asistentes;
        j = j + 1;
        asistentes = j;
    }
    return NULL;
}

int main() {
    pthread_t p1, p2;
    pthread_create(&p1, 0, puerta1, NULL);
    pthread_create(&p2, 0, puerta2, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("Cantidad de asistentes (debe ser 2000) %d\n", asistentes);
    return 0;
}
```

Instituto Politécnico Superior

Pthreads: Sincronización

- Existen diversas herramientas para sincronizar pthreads

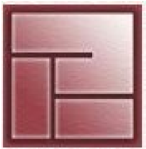
- **pthread_join**
  Suspende la ejecución de un thread hasta que otro termine

- **pthread_once**
  Sirve para asegurar que las rutinas de inicio se ejecutan una única vez cuando son invocadas por múltiples threads

- **pthread_mutex** - Semáforos binarios (mutex)
  Facilita el control de acceso ordenado (exclusivo) a recursos compartidos

- **pthread_cond_v** - Variables de Condición.
  Permiten que un thread se "suscriban" a un evento (contador alcanzando un valor, un flag cambiando, etc). Tal evento indica que cierta condición se alcanzó y esto permite la continuación de la ejecución del thread

- *A partir de estas herramientas se pueden construir elementos más complejos como ser read/write exclusión, semáforos n-arios*



pthread_mutex

- El tipo **`pthread_mutex_t`**
- Uso
  - Crear e inicializar un mutex para cada recurso a proteger

Inicializar:

Estáticamente

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Dinámicamente

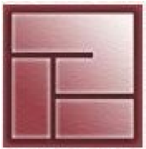
```
int phtread_mutex_init(pthread_mutex_t *,
                       const pthread_mutex_attr_t *)
pthread_mutex_t *mutex;
mutex = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t))
ptrhead_mutex_init(mutex, NULL);
```

- Antes de acceder al recurso se debe invocar

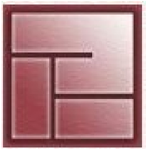
```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- Luego de usar el recurso invocar

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




```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
int asistentes = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* puerta1(void *x) {
   int i;
   int k;
   for(k=0; k<1000; k++) {
       pthread_mutex_lock(&mutex);
       i = asistentes;
       i = i + 1;
       asistentes = i;
       pthread_mutex_unlock(&mutex);
   }
   return NULL;
}

void* puerta2(void *x) {
   int j;
   int k;
   for(k=0; k<1000; k++) {
       pthread_mutex_lock(&mutex);
       j = asistentes;
       j = j + 1;
       asistentes = j;
       pthread_mutex_unlock(&mutex);
   }
   return NULL;
}

int main() {
    pthread_t p1, p2;
    pthread_create(&p1, 0, puerta1, NULL);
    pthread_create(&p2, 0, puerta2, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Cantidad de asistentes (debe ser 2000) %d\n", asistentes);
    return 0;
}
```



Variables de Condición

- Tipo **pthread_mutex_cond**
- Las variables condición siempre deben estar asociadas a un mutex a fin de evitar race conditions

- Soportan tres tipos de operaciones:

wait: suspende al proceso que la invoca en la condición

`pthread_cond_wait`

signal: activa un proceso suspendido en la condición

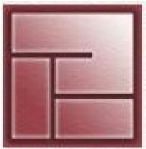
`pthread_cond_signal`

broadcast: activa a todos los procesos suspendidos en la condición

`pthread_cond_broadcast`

Existe un método para limitar el tiempo que un hilo está bloqueado en una variable condición

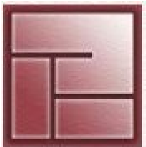
`pthread_cond_timedwait`



Variables de Condición

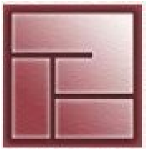
- Funciones de creación/destrucción

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;


int pthread_cond_destroy(pthread_cond_t *cond)
```



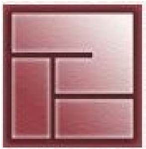
Variables de Condición

Llamada	Descripción
pthread_cond_wait(cond, mut)	De forma atómica, realiza la operación p_thread_mutex_unlock sobre *mut*, y bloquea al hilo en la variable condición *cond*. Cuando es despertado, el hilo cierra nuevamente el mutex *mut* (realizando la operación p_thread_mutex_lock.
pthread_cond_signal(cond)	Despierta a uno de los hilos que están bloqueados en la variable condición. Si no hay hilos bloqueados, no sucede nada.
pthread_cond_broadcast(cond)	Despierta todos los hilos bloqueados sobre *cond*.
pthread_cond_timedwait (cond, mut,duracion)	Igual que pthread_cond_wait pero si antes de *duracion* no se ha despertado al hilo, la llamada finalizará con un código de error. Al despertar, el hilo que invoca la llamada, vuelve a cerrar el mutex *mut*.



- Variables de Condición
  - Productor Consumidor

```
#define n 10

int entrada, salida, contador;
int buffer[n];

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t lleno = PTHREAD_COND_INITIALIZER;
pthread_cond_t vacio = PTHREAD_COND_INITIALIZER;
```



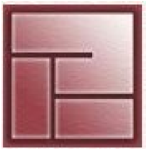
Variables de Condición

- **Productor Consumidor**

```
void *productor(void * arg){
  int i;
  for (i=0; i<100; i++)
    Insertar(i);
  pthread_exit(0);
}
```

```
void *consumidor(void * arg){
  int i;
  for (i=0; i<100; i++)
    Extraer();
  pthread_exit(0);
}
```

```
int main() {
  pthread_t th_a, th_b;
  entrada = salida = contador = 0;

  pthread_create(&th_a, NULL, productor, NULL);
  pthread_create(&th_b, NULL, consumidor, NULL);
  pthread_join(th_a, NULL);
  pthread_join(th_b, NULL);

  exit(0);
}
```



Variables de Condición

- Productor Consumidor

```
Insertar (int dato) {
  pthread_mutex_lock(&mutex);
  while (contador >= n) pthread_cond_wait(&lleno, &mutex);
  buffer[entrada]= dato;
  printf("--> Se produce %d\n", dato);
  entrada = (entrada+1) % n;
  contador = contador + 1;
  pthread_cond_broadcast(&vacio);
  pthread_mutex_unlock(&mutex);
}
```

```
Extraer () {
  int dato;
  pthread_mutex_lock(&mutex);
  while (contador == 0) pthread_cond_wait(&vacio, &mutex);
  dato = buffer[salida];
  printf("<-- Se consume %d\n", dato);
  salida = (salida+1) % n;
  contador= contador - 1;
  pthread_cond_broadcast(&lleno);
  pthread_mutex_unlock(&mutex);
}
```