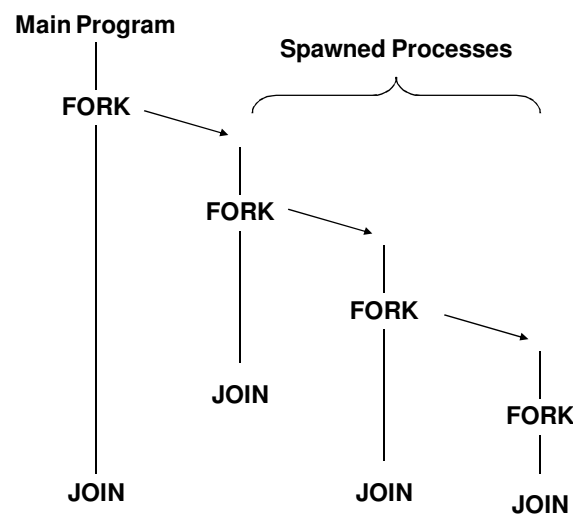


Shared Address Space Computing: Programming

Alistair Rendell

See Chapter 6 or Lin and Synder, Chapter 7 of Grama,
Gupta, Karypis and Kumar, and Chapter 8 of Wilkinson
and Allen

Fork/Join Programming Model



(Heavyweight) UNIX Processes

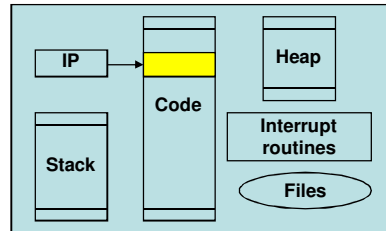
- O/S like UNIX is based on the notion of a process
 - The CPU is shared between different processes
- UNIX processes created via `fork()`
 - Child copy an exact copy of parent, except it has unique process ID
- Processes are “joined” using the system calls `wait()`
 - This introduces a synchronization

UNIX Fork Example

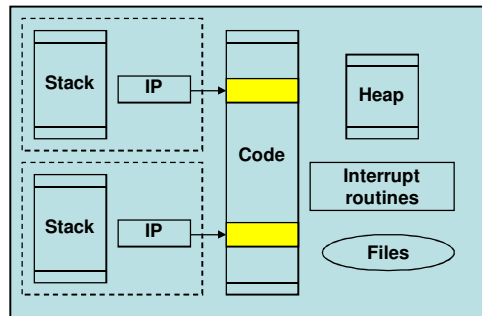
```
Pid = fork();  
If (pid == 0) {  
    code to be executed by slave  
} else {  
    code to be executed by parent  
}  
If (pid == 0) exit (0); else wait(0);
```

- *What is a zombie process?*
- *Why might fork be viewed as a heavyweight operation?*
 - *How is this cost reduced?*

Processes and Threads



A process

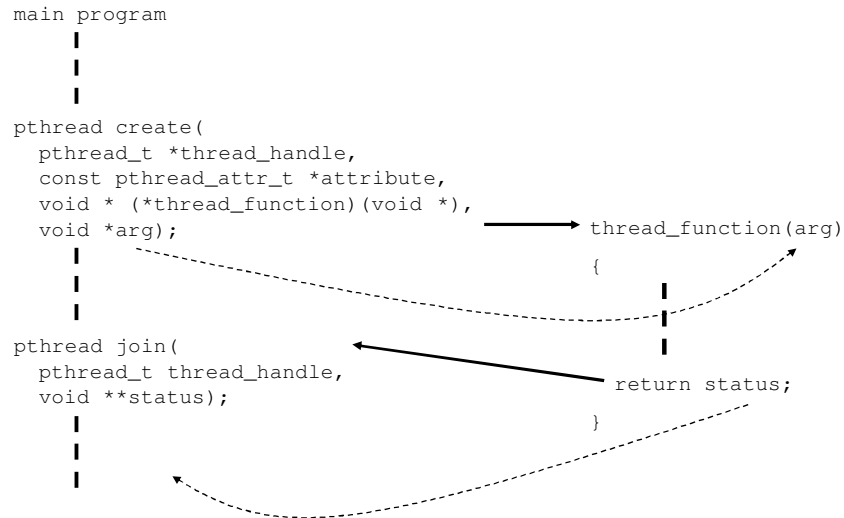


Two threads

Why Threads

- Software Portability
 - Applications can be developed on serial machine and run on parallel machines without changes (*is this really true?*)
- Latency Hiding
 - Ability to mask access to memory, I/O or communication by having another thread execute in the meantime (*but how quickly can execution switch between threads?*)
- Scheduling and Load Balancing
 - For unstructured and dynamic applications (e.g. game playing) loading balancing can be very hard. One option is to create more threads than CPU resources and let the O/S sort out the scheduling
- Ease of Programming, Widespread Use
 - Due to above, threaded programs are easier to write (or develop incrementally), so there has been widespread acceptance of POSIX thread API (generally referred to as pthreads)

Pthread Creation



Threads and Threaded Code

- `pthread_self()` provides ID of calling routine
 - `pthread_equal(thread1, thread2)` tests ID
 - *What is the analogous MPI call?*
- **Detached Threads**
 - Threads that will never synchronize via a join
 - Specified via an attribute
 - *Why should we care?*
 - *How do you handle this with UNIX fork?*
- **Re-entrant** or **thread-safe** routines are those than can be safely called when another instance has been suspended in the middle of its invocation
 - *Can you give an example of when a routine might not be re-entrant?*

Example: Computing Pi

```

#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}

```

Example: Computing Pi (cont)

```

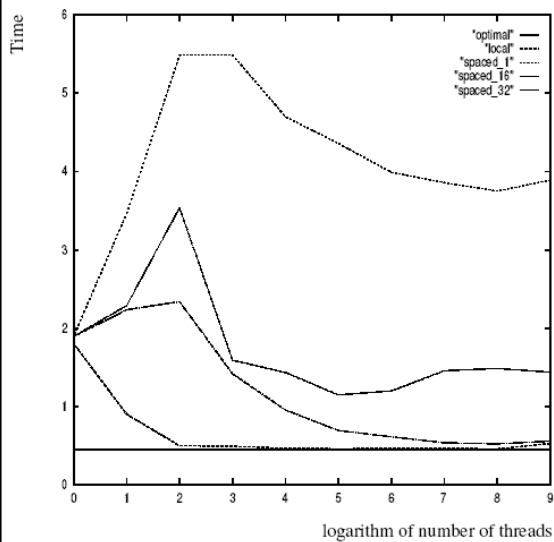
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double) (rand_r (&seed)) / (double) ((2<<14)-1);
        rand_no_y = (double) (rand_r (&seed)) / (double) ((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}

```

Programming and Performance Notes

- Note the use of the function `rand_r` (instead of superior random number generators such as `drand48`).
- Executing this on a 4-processor SGI Origin, we observe a 3.91 fold speedup at 32 threads. This corresponds to a parallel efficiency of 0.98!
- We can also modify the program slightly to observe the effect of false-sharing.
- The program can also be used to assess the secondary cache line size.

Performance



- 4 processor SGI Origin System using upto 32 threads
- Instead of incrementing `local_hits`, we add to a shared array using stride of 1, 16 and 32
- (Taken from Grama, Gupta, Karypis and Kumar reference)

Sharing Data

- Shared memory provides ability to share data directly without having to pass messages
- For UNIX heavyweight processes additional shared memory system calls are necessary
 - Each process has its own virtual address space
 - Shared memory system calls allow the processes to “attach” to a segment of shared memory
 - Use `shmget()` to allocate a shared memory segment, `shmat()` to attach to it (either at a specific virtual address or allow the O/S to return a suitable address)
- For threads, variables on the heap are shared
 - *What variables reside on the heap?*
 - *What variables reside on the stack?*

Accessing Shared Data

- Concurrent read is no problem
 - but what about concurrent write?

	Instruction	P0	P1
Time ↓	<code>x=x+1</code>	read x compute x+1 write x	read x compute x+1 write x

- Similar problems with access to shared resources like I/O
- **Critical Sections:** mechanism to ensure controlled access to shared resources
 - Processes enter critical section under **mutual exclusion**

Locks

- Simplest mechanism for providing mutual exclusion
- A lock is a 1-bit variable, a value of
 - 1 indicates a process is in the critical section
 - 0 indicates no process is in the critical section
- At its lowest level a lock is a protocol for coordinating processes,
 - the CPU is not physically prevented from executing those instruction

```
while (lock == 1) do_nothing;    /*infinite testing loop */
lock = 1;                       /*enter critical section*/
critical section
lock = 0;                       /*exit critical section*/
```

- The above is an incorrect example of a **spin-lock**, that uses a mechanism called **busy waiting**
 - *What is wrong with the above?*
 - *What low level hardware support is provided to address this?*

Pthread Lock Routines

- Locks implemented as mutually exclusive lock variables or *mutex* variables
- To use a variable must be declared as type `pthread_mutex_t`
 - Usually in main thread as it needs to be visible to all threads using it

```
pthread_mutex_t mutex1;          (NULL specifies default
pthread_mutex_init(&mutex1, NULL); attributes for mutex)

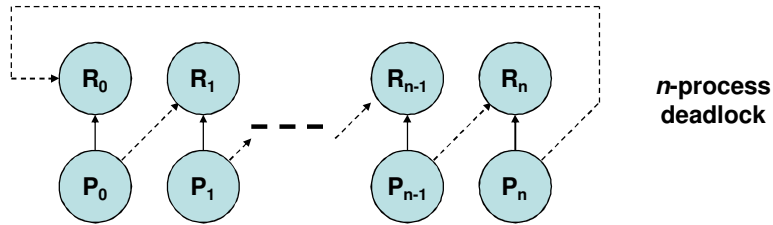
pthread_mutex_lock(&mutex1);

/* CRITICAL SECTION */

pthread_mutex_unlock(&mutex1);
```

Deadly Embrace!

- **Deadly embrace** or **deadlock** can occur if two (or more) processes are attempting to acquire two (or more) resources that cannot be shared
 - *How does this relate to a group of dining philosophers?*



- Pthreads provides `pthread_mutex_trylock()` to help address these circumstances

Semaphores#1

- Devised by Dijkstra (1968)
- A positive integer (including zero) operated on by two operations named P (passeren) and V (vrijgeven)
 - P(s) waits until s is greater than 0 and decrements s by 1 and allows the process to continue
 - V(s) increments s by 1 to release any one of the waiting processes (if any)
 - P and V operations are indivisible (atomic)
- In following s initialized to 1

P0	P1	P2
Noncritical section	Noncritical section	Noncritical section
⋮	⋮	⋮
P(s)	P(s)	P(s)
Critical Section	Critical Section	Critical Section
V(s)	V(s)	V(s)
⋮	⋮	⋮
Noncritical section	Noncritical section	Noncritical section

Semaphores#2

- Binary semaphores only a value of 0 or 1
 - hence act as a lock
- UNIX provides semaphores IPC (inter-procedural communication) via
 - `semget()` to get a semaphore array
 - `semctl()` to set initial value
 - `semop()` to perform operations
- pthreads does not provide semaphores
 - Easy to construct
 - Available in real-time extensions
- *What's bad about semaphores?*

Monitors (briefly)

- Hoare (1974): a higher level technique for implementing mutual exclusion
 - A suite of routines that provide the only method to access a shared resource
 - Essentially data and operations on that data are encapsulated into one structure
- A monitor can only have one active instruction pointer executing instructions from within the monitor at any one time
 - *Why active?*
- The concept of a monitor exists within Java through the `synchronized` keyword
- Not available in pthreads or for UNIX processes
 - A similar concept can be constructed

Condition Variables#1

- A piece of code that needs to be executed when a certain global condition exists
 - E.g when a certain value of a variable is reached
- Using a lock would require frequent *polling* of the lock value
 - This would be *busy waiting*
- Condition variables define three operations
 - wait(cond_var) -wait for condition to occur
 - signal(cond_var) -signal that condition occurred
 - status(cond_var) -No of procs waiting on condition
- Wait will also release a lock or semaphore and can be used to allow another process to alter the condition

Condition Variable Example

- Consider example where process 0 will take some action when the value of a global counter becomes zero
 - In the following wait unlocks the lock and suspends the process
 - The while statement will re-check the condition and is for good error checking

Process 0	Other Processes
<pre> action() { : lock(); while (x != 0) wait(s); ← unlock(); take_action(); : } </pre>	<pre> counter() { : lock(); x--; if (x == 0) signal(s); unlock(); : } </pre>

pthread Condition Variables

- Associated with a specific mutex
- Creation routines

```
pthread_cond_t cond1;
pthread_cond_init(&cond1, NULL);
pthread_cond_destroy();
```

- Waiting routines

```
pthread_cond_wait(cond1, mutex1);
pthread_cond_timedwait(cond1, mutex1, abstime);
```

- Signaling routines

```
pthread_cond_signal(cond1);
pthread_cond_broadcast(cond1);
```

– Signals are not remembered (*what does this mean?*)

pthread CV Example

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
:
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
:
pthread_create()
:
```

Process 0

```
action()
{
:
pthread_mutex_lock(&mutex1);
while (x != 0)
    pthread_cond_wait(cond1, mutex1);
pthread_mutex_unlock(&mutex1);
take_action();
:
}
```

Other Processes

```
counter()
{
:
pthread_mutex_lock(&mutex1);
x--;
if (x == 0)
    pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
:
}
```

Barriers

- As with Message Passing, process/thread synchronization often required
 - Pthreads does not provide a native barrier so it must be coded
 - *See laboratory for one example of how this might be done*

Concurrent v Parallel Processing

- **Concurrent Processing:** Environment in which tasks are defined and allowed to execute in any order
 - Does not imply a multiple processor environment
 - Eg spawn a separate thread to do I/O while CPU intensive task continues to do another operation
- **Parallel Processing:** The simultaneous execution of concurrent tasks on different CPUs

All parallel processing is concurrent, but NOT all concurrent processing is parallel

O/S Thread Support

- O/S Originally designed for process not thread support
- Require O/S to
 - Treat threads equally and ensure that they all get equal (or user defined) access to machine resources
 - Know what to do when a thread issues a fork command
 - Be able to deliver a signal to the correct thread
- Libraries executed by a multi-threaded program need to be thread safe
 - Potential static data structures to be overwritten
- Hardware support
 - Some hardware provide support for very fast context switch between threads, e.g. Cray MTA or more recently hyper-threading

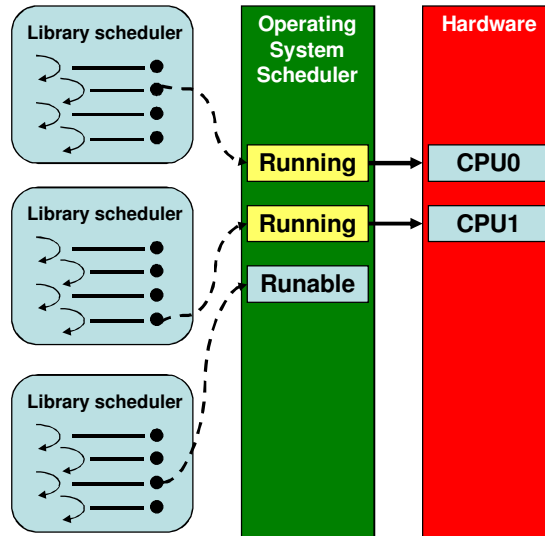
Thread Implementations

- Three categories
 - Pure user space
 - Pure kernel space
 - Mixed
- User Mode: When a process executes instructions within its program or linked libraries
- Kernel Mode: When the operating system executes instructions on behalf of a user program
 - Often as a result of a system call
 - In kernel mode the program can access kernel space

User Space Threads

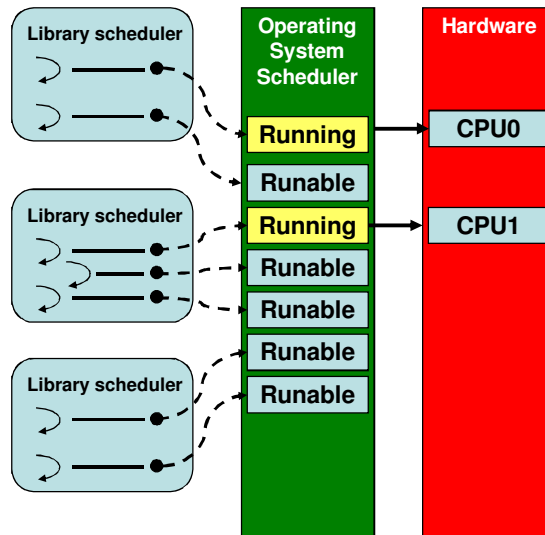
- No kernel involvement in providing a thread
- Kernel has no knowledge of threads and continues to schedule processes only
- Thread library selects which thread to run

OK for concurrency, no good for parallel programming



Kernel Space Threads

- Kernel level thread created for each user thread
- O/S must manage on a per-thread basis information typically managed on a process basis
- Potentially poor scaling when too many threads as O/S gets overloaded



User-space v Kernel-space

- User Space Advantages
 - No changes to core O/S
 - No kernel involvement means they may be faster for some applications
 - Can create many threads without overloading the system
- User Space Disadvantages
 - Potentially long latency during system service blocking (e.g. 1 thread stalled on I/O request)
 - All threads compete for the same CPU cycles
 - No advantage gained from multiple CPUs
- Mixed scheme
 - A few user threads mapped to one kernel thread