Operating Systems

Threads

Chapter: Threads: Questions

- How is a thread different from a process?
- Why are threads useful?
- How can POSIX threads be useful?
  - (Portable Operating System Interface) API – enabling portability between UNIX(es) and other operating systems.
- What are user-level and kernel-level threads?
- What are problems with threads?

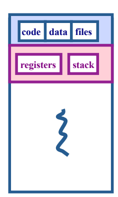
Review: What is a Process?

A process is a program in execution…

A thread have
(1) an execution stream and
(2) a context

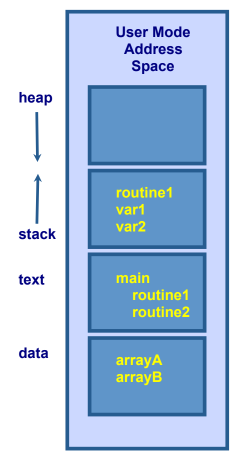
- Execution stream
  - stream of instructions
  - sequential sequence of instructions
  - "thread" of control
- Process 'context' (seen picture of this already)
  - Everything needed to run (restart) the process …
  - Registers
    - program counter, stack pointer, general purpose…
  - Address space
    - Everything the process can access in memory
    - Heap, stack, code

Running on a thread



Review: What Makes up a Process?

- Program code (text)
- Data
  - global variables
  - heap (dynamically allocated memory)
- Process stack
  - function parameters
  - return addresses
  - local variables and functions
- OS Resources
- Registers
  - program counter, stack pointer



address space are the shared resources of a(ll) thread(s) in a program

Issues with Processes()?

- How do processes (*independent* memory space) *communicate*?
  - Not really that simple (seen it, tried it – and you have too):
    - Message passing (send and receive)
    - Shared Memory: Set up a shared memory area (easier)?
- Problems:
  - Overhead: Both methods add some kernel overhead lowering (potential) performance
  - Complicated: IPC is not really that "natural"
    - increases the complexity of your code

```
main()
   {
   i = 55;
   fork();
   // what is i
```


Processes versus Threads

Thread (s):

- An execution stream that shares an address space
  - Overcome data flow over a file descriptor
  - Overcome setting up `tighter memory' space
- Multiple threads within a single process

```
main()
   {
   i = 55;
   fork();
   // what is i
```

Examples:

- Two processes (copies of each other) examining memory address **`0xffe84264`** see different values (i.e., different contents)
  - same frame of reference
- Two threads examining memory address **`0xffe84264`** see same value (i.e., same contents)
- Examples:
  - **`ctest/i-process.c`**
  - **`ctest/i-threading.c,`**

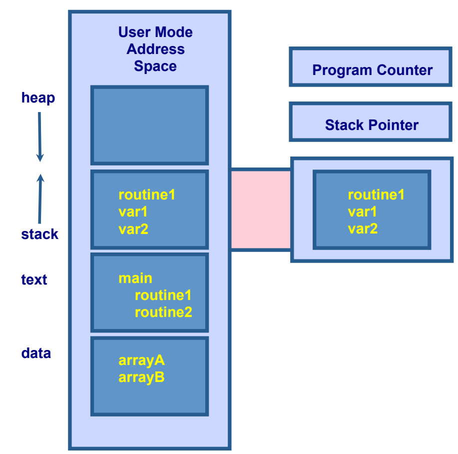
```
thread: shares i same memory storage
```



```
#include <stdio.h>   // # printf
#include <unistd.h>  // # fork
#include <stdlib.h>  // # exit

pid_t childpid = -1;
int i;

int main( int argc, char *argv[] )
{
 int i = -55;

 if( (childpid = fork()) == 0 )
        {
        fflush(stdout);
        printf("[1. child  (%10d)]: i = %3d address = %p\n", childpid, i, (void*) &i );
        sleep(1);
        i = 11;
        printf("[2. child  (%10d)]: i = %3d address = %p\n", childpid, i, (void*) &i );
        exit(0);
        }
  else
        {
        // try to make sure parent is executed after child 'changes' i.
        sleep(10);
        printf("[b. parent (%10d)]: i = %3d address = %p\n", childpid, i, (void *) &i );
        }
  wait( (int *) 0 );

  printf("[w. parent (%10d)]: i = %3d address = %p\n", childpid, i, (void *) &i );
}
```

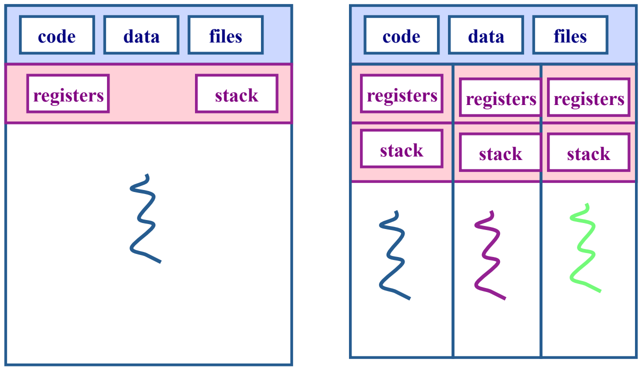

What Makes up a Thread?

- **Own stack (Is it necessary?)**
- **Own registers (Is it necessary?)**
  - » **Own program counter**
  - » **Own stack pointer**
- **State (running, sleeping)**
- **Signal mask**



address space are the shared resources of a(ll) thread(s) in a program

Single and Multithreaded Process



Why Support Threads?

- *Divide large task across several cooperative threads*
- Multi-threaded task has many performance benefits

- **Examples:**
  - » **Web Server: create threads to:**
    - – **Get network message from client**
    - – **Get URL data from disk**
    - – **Compose response**
    - – **Send a response**
  - » **Word processor: create threads to:**
    - – **Display graphics**
    - – **Read keystrokes from users**
    - – **Perform spelling and grammar checking in background**

Why Support Threads?

- Divide large task across several cooperative threads
- *Multi-threaded task has many performance benefits*

- **Adapt to slow devices**
  - » **One thread waits for device while other threads computes**
- **Defer work**
  - » **One thread performs non-critical work in the background, when idle**
- **Parallelism**
  - » **Each thread runs simultaneously on a multiprocessor**

Why Threads instead of a Processes?

- Advantages of Threads:
  - – Thread operations cheaper than corresponding process operations
    - In terms of: Creation, termination, (context) switching
  - – IPC cheap through shared memory
    - No need to invoke kernel to communicate between threads
- Disadvantages of Threads:
  - – **True** Concurrent programming is a challenge (what does this mean? True concurrency?)
  - – Synchronization between threads needed to use shared variables (more on this later – this is HARD).
  - – Parallelism vs. Concurrency

Why are Threads Challenging?

pthread1 Example: Output?

```
gcc pthread1.c -o pthread1 -lpthread
```

```
main()
{
   pthread_t t1, t2;
   char *msg1 = "Thread 1"; char *msg2 = "Thread 2";
   int ret1, ret2;
   ret1 = pthread_create( &t1, NULL, print_fn, (void *)msg1 );
   ret2 = pthread_create( &t2, NULL, print_fn, (void *)msg2 );
   if( ret1 || ret2 )
   {
        fprintf(stderr, "ERROR: pthread_created failed.\n");
        exit(1);
   }
   pthread_join( t1, NULL );
   pthread_join( t2, NULL );
   printf( "Thread 1 and thread 2 complete.\n" );
}
void print_fn(void *ptr)
{
   printf("%s\n", (char *)ptr);
}
```


Why are Threads Challenging?

- Example: Transfer $50.00 between two accounts and output the total balance of the accounts:

M = **Balance in Maria's account (begin $100)**

T = **Balance in Tucker's account (begin $50)**

B = **Total balance**

- Tasks:

```
T = 50, M = 100
M = M - $50.00
T = T + $50.00
B = M + T
```

```
Idea: on distributing
the  tasks:
(1)One thread debits
   and credits
(2) The other Totals
Does that work?
```


Why are Threads Challenging?

- Tasks:

```
T = 50, M = 100
M = M - $50.00     } One thread debits
T = T + $50.00       & credits
B = M + T          } One thread totals
```

M = M - $50.00	M = M - $50.00	B = M + T
T = T + $50.00	B = M + T	M = M - $50.00
B = M + T	T = T + $50.00	T = T + $50.00
B = $150	B = $100	B = $150

Common Programming Models

- Manager/worker
  - Single manager handles input and assigns work to the worker threads
- Producer/consumer
  - Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- Pipeline
  - Task is divided into series of subtasks, each of which is handled in series by a different thread

Thread Support

- Three approaches to provide thread support
  - User-level threads
  - Kernel-level threads
  - Hybrid of User-level and Kernel-level threads

Latencies

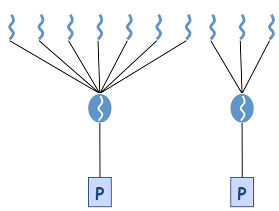
- Comparing user-level threads, kernel threads, and processes
- Thread/Process Creation Cost: Null fork
  - **Evaluate** –with Null fork: the time to create, schedule, execute, and complete the entity that invokes the null procedure
- Thread/Process Synchronization Cost: Signal-wait
  - **Evaluate** – with Signal-Wait: the time for an entity to signal a waiting entity and then wait on a condition (**overhead of synchronization**)

Procedure call = 7 us Kernel Trap = 17 us	User Level Threads	Kernel Level Threads	Processes
Null fork	34	948	11,300
Signal-wait	37	441	1,840

30X,12X

User-Level Threads

- Many-to-one thread mapping
  - Implemented by user-level runtime libraries
    - Create, schedule, synchronize threads at user-level, state in user level space
  - OS is not aware of user-level threads
    - OS thinks each process contains only a single thread of control



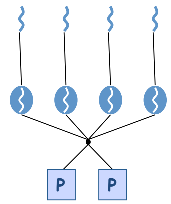
- **Advantages**
  - » **Does not require OS support; Portable**
  - » **Can tune scheduling policy to meet application (user level) demands**
  - » **Lower overhead thread operations since no system calls**
- **Disadvantages**
  - » **Cannot leverage multiprocessors (no true parallelism)**
  - » **Entire process blocks when one thread blocks**

Blocked UL Threads: Jacketing

- Avoids 'blocking' on system calls that block (e.g., I/O)
- Solution:
  - Instead of calling a blocking system call call an application level I/O jacket routine (a nonblocking call)
  - Jacket routine provides code that determines whether I/O device is busy or available (idle).
  - Busy:
    - Thread enters the ready state and passes control to another thread
    - Control returns to thread it retries
  - Idle:
    - Thread is allowed to make system call.

Kernel-Level Threads

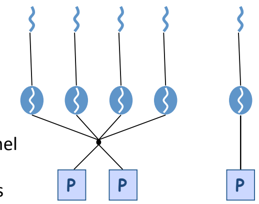
- One-to-one thread mapping
  – OS provides each user-level thread with a kernel thread
  – Each kernel thread scheduled independently
  – Thread operations (creation, scheduling, synchronization) performed by OS



- **Advantages**
  » **Each kernel-level thread can run in parallel on a multiprocessor**
  » **When one thread blocks, other threads from process can be scheduled**
- **Disadvantages**
  » **Higher overhead for thread operations**
  » **OS must scale well with increasing number of threads**

Two-Level Model

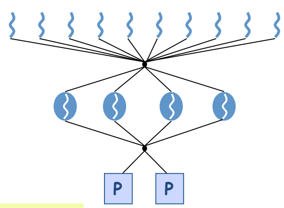
- one-one & (strict) many-to-many
  – OS provides each user-level thread with a kernel thread
  – Supports both bound an unbound threads
    - Bound threads - permanently bound to a single kernel level thread
    - Unbound threads may move to other kernel threads



- **Advantages**
  » **Flexible, best of two worlds**
- **Disadvantages**
  » **More complicated**

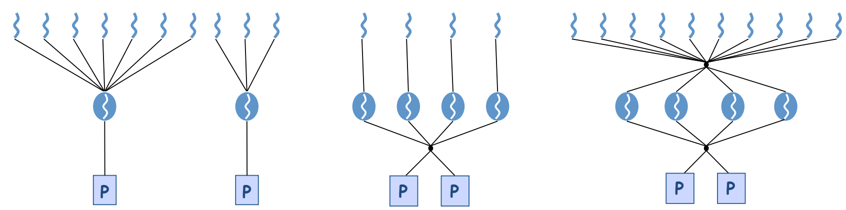
Hybrid of Kernel & User -Level Threads

- m - n thread mapping (many to many)
  – Application creates m threads
  – OS provides *pool* of n kernel threads
  – Few user-level threads mapped to each kernel-level thread



- **Advantages**
  » **Can get best of user-level and kernel-level implementations**
  » **Works well given many short-lived user threads mapped to constant-size pool**
- **Disadvantages**
  » **Complicated…**
  » **How to select mappings?**
  » **How to determine the best number of kernel threads?**
    – **User specified**
    – **OS dynamically adjusts number depending on system load**

Summary: Thread Models



- Kernel Level: Windows 95/98/NT/2000, Solaris, Linux
- User Level: Mach, C-threads, Solaris threads
- Hybrids: IRIX, HP-UX, True 64 UNIX, Older Solaris models
- API: POSIX P-threads
  – → Native threading interface for Linux now 1:1 model

Design: Threading Issues: fork() & exec()

- fork()
  - Duplicate all threads?
  - Duplicate only the thread that performs the fork
  - Resulting new process is single threaded?
  - -> solution provide two different forks (mfork)
- exec()
  - Replaces the process - including all threads?
  - If exec is after fork then replacing all threads is unnecessary.

Threading Issues: Cancellation

- Example 1: User pushes top button on a web browsers - while other threads are images (one thread per image).
  - Asynchronous Cancellation: Immediate (OS need to reclaim resources)
- Example 2: Several threads concurrently searches data base and one thread finds target data.
  - Deferred Cancellation: Thread terminates it self when notices it is scheduled for termination.

Threading Issues: Threads and Signals

- Problem: To which thread should OS deliver signal?
- Option 1: Require sender to specify thread ID (instead of process id)
  - Sender may not know about individual threads
- Option 2: OS picks destination thread
  - POSIX: Each thread has signal mask (disable specified signals)
  - OS delivers signal to all threads without signal masked
  - Application determines which thread is most appropriate for handing signal
- Synchronous - delivered to the same process that caused the signal
- Asynchronous - event is external to running process.

Other Thread Issues

- Creating thread is still costly…
- No bound of number of threads…

Thread Pools

- Create a number of threads in a pool where a number of threads await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than waiting to create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- The number of threads can be set heuristically based on the hardware and can even be dynamically adjusted taking into account user statistics.

IPC: Shared Memory

- Processes
  - Each process has private address space
  - Explicitly set up shared memory segment within each address space
- Threads
  - Always share address space (use heap for shared data), don't need to set up shared space already there.
- Advantages
  - Fast and easy to share data
- Disadvantages
  - Must *synchronize* data accesses; error prone (later)

IPC: Message Passing (also for threads, similar to processes)

- Message passing most commonly used between processes
  - Explicitly pass data between sender (src) + receiver (destination)
  - Example: Unix pipes
- Advantages:
  - Makes sharing explicit
  - Improves modularity (narrow interface)
  - Does not require trust between sender and receiver
- Disadvantages:
  - Performance overhead to copy messages
- Issues:
  - How to name source and destination?
    - One process, set of processes, or mailbox (port)
  - Does sending process wait (I.e., block) for receiver?
    - Blocking: Slows down sender
    - Non-blocking: Requires buffering between sender and receiver

IPC: Signals

- Signal
  - Software interrupt that notifies a process of an event
  - Examples: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT
- What happens when a signal is received?
  - Catch: Specify signal handler to be called
  - Ignore: Rely on OS default action
    - Example: Abort, memory dump, suspend or resume process
  - Mask: Block signal so it is not delivered
    - May be temporary (while handling signal of same type)
- Disadvantage [signals]
  - Does not specify any data to be exchanged
  - Complex semantics with threads

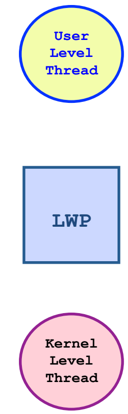
*** Board & Read

Scheduler Activations (Notes)

- Provides better OS support for user level threading
  - Dynamic adjustment of number of kernel level threads to user level threads:
    - E.g. Two level and the m:n thread models need to maintain appropriate ratios
  - Key Idea: Kernel notifies thread scheduler of all kernel events via
    - up-calls()

Scheduler Activations

- Use an intermediate data structure between user/kernel level threads.
- Details: User level threads run and are scheduled (by the user level scheduler) on 'virtual processor'
  - A data structure or light-weigh process (LWP) that is between the kernel thread and the user thread.
  - Each LWP is attached to a kernel thread and kernel threads are what the OS schedules to run on physical processors.



Scheduler Activations

- An application may require any number of LWPs to run efficiently.
  - Example: A CPU-bound application on a single processor.
    - Needs only one LWP.
  - Example: An I/O-bound application
    - May need many LWPs- one for each concurrent blocking system since if there are not enough LWPs, the unassigned threads must wait for one of the LWPs to return from the kernel.

Scheduler Activations (notes)

- Why not a user level thread scheduler that spawns a kernel thread for blocking operations?
  - Forget spawning, use a pool of kernel threads.
  - But how do we know if an operation will block?
    - read might block, or data might be in page cache.
    - Any memory reference might cause a page fault to disk.
- Scheduler Activations
  - Kernel tells user when a thread is going to block, via an upcall.
  - Kernel can provide a kernel thread to run the user-level upcall handler (or preempt user thread).
  - User-level scheduler suspends blocking thread and can give back kernel thread it was running on.

Quiz 3

1. What resources (context) within a process **are shared** between threads?
2. What resources (context) **cannot be shared** among threads within the same process?
3. What happens to other p-threads within the same process when a thread reads from disk?
4. Are POSIX threads user OR kernel level threads ?
5. Do Java threads use kernel or user level threads (Justify your answer)