2-3 Threads

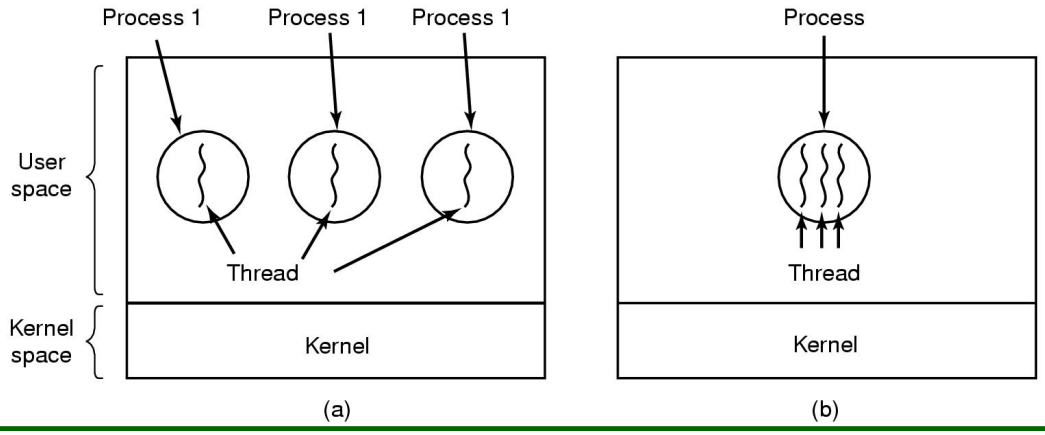
Stallings: 4.1, 4.2, 4.3, 4.6

Introduction to Threads

- The overhead of using multiple "full-blown" processes can limit the performance of many classes of applications (e.g., server applications)
- The concept of threads, a.k.a. lightweight processes, came under study in the 1980s.
- Now, many commercial and public-domain thread packages are available and are widely used.
- Processor architectures also support threads
- Multi-threaded programming is often challenging
- In many thread packages, semantics is closely tied to the operating system…

Definition

- Thread: execution of a sequence of instructions within an address space
- A process can contain one or more threads
- Multiple threads wandering through the same program concurrently, maybe simultaneously!

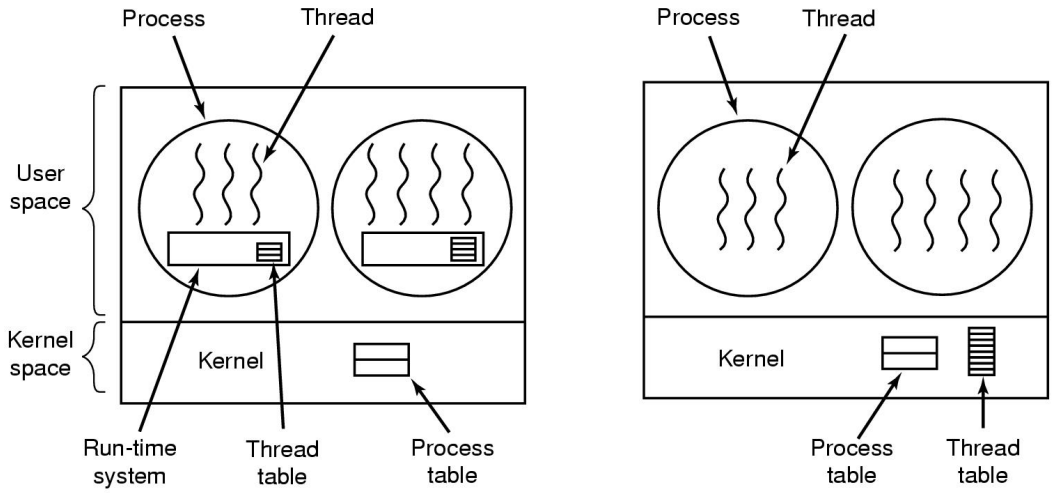


User-level vs. Kernel-level Threads

- Threads may, or may not, be supported by the operating system
- User-level threads
  - Thread support implemented within one user-level process
  - Part of the user-level program (run-time system) manages and schedules the threads, all within one process
  - Threads are **not** known to the kernel
- Kernel-level threads
  - Threads are known to kernel
  - State information (like PCB) maintained on thread basis
  - Usually can be scheduled independently

User-Level vs. Kernel-Level Support

- ULT: kernel sees only the **process** (one thread)
- KLT: kernel maintains info on each thread



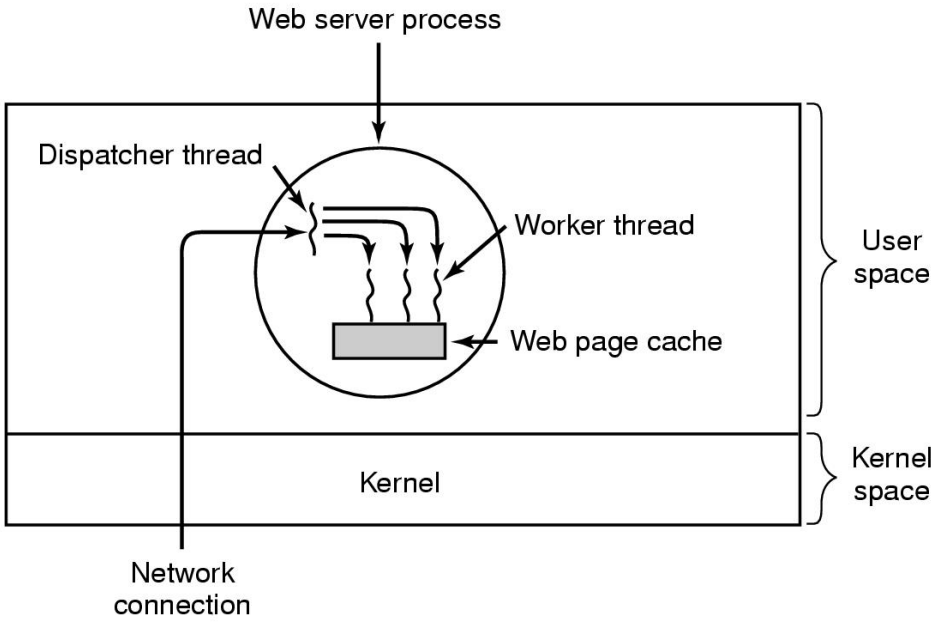
- Implications for concurrency? performance?

Multi-threaded programming

- Program is written as a set of threads
- The threads execute concurrently within single process address space.
- Threads are designed to cooperatively implement some functionality. E.g.,
  - web server
  - parallel processing application
  - parallel database application
- Threads must interact via synchronization primitives
  - E.g., semaphores, mutexes, condition variables, etc.
  - Otherwise? inconsistent access to shared data

Example of Thread Usage

A multithreaded Web server.



Simplified code…

- (Note: Does not show the synchronization primitives)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

"dispatcher" thread

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

"worker" thread

Thread Advantages

- Performance: interactions and switches among threads is faster than for processes
- Even when executed on a uniprocessor, threads are often seen as a more "natural" way to program certain classes of applications:
  - Servers. How so?
  - Simulation. How so?
  - Pipelined data processing (producer/consumer)

Can be scheduled to run in multiprocessors

- Kernel-level threads can "automatically" take advantage of a multi-processor/core system.

Parallel Speedup

- Speedup for P processors:
  - $S_N = T_1/T_N$, where T_1 is serial time, T_N is time on N processors
- Efficiency, $E_N = S_N/P = T_1/(P*T_N)$
- Example:
  - $T_1 = 12s$, $T_4 = 4s$
  - $S_4 = ?$ 12/4 = 3
  - $E_4 = ?$ 12/(4*4) = 12/16 = 0.75

Amdahl's Law

- Limit on speedup depends on the fraction of code that can (and cannot) be parallelized
- $S_N = T_1/T_N < 1/((1-f) + f/N))$, where
  - f is the fraction of code that can be parallelized

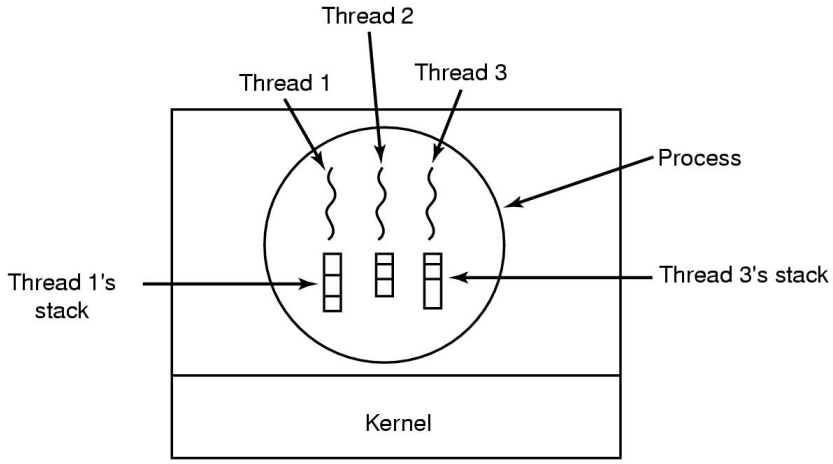
- Example: f = 0.9
  - $S_4 <$ 1/(0.1+0.9/4) = 3.1
  - $S_{10} <$ 1/(0.1+0.9/10) = 5.3
  - $S_{100} <$ 1/(0.1+0.9/100) = 9.2
  - $S_{1000} <$ 1/(0.1+0.9/1000) = 9.9

Multiprocessor Thread Performance

- Thread-based applications may be executed on either uniprocessors or multiprocessors
- On a multiprocessor, performance depends on the level of OS support:
  - User-level threads? max speed: 1.0
  - Kernel-level threads >= 1.0

Each Thread Needs…

- Whether implemented as kernel- or user-level threads
  - Clearly, each thread needs its own stack.
  - Per-thread global variables are highly desirable.



Thread-Specific and Shared Resources

- Each thread has its own context:
  - program counter, stack, register set, etc.
- In addition to address space, all threads within a single process typically share:
  - open files, signals and other kernel services

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Thread/Process Issues

- Semantics of the interaction with the kernel can complicate programming
  - Has been a major issue in Linux over the years…
- Examples:
  - When a thread invokes fork(), are all threads instantiated in the new process?
  - What if one thread is blocked waiting for input from a keyboard? Are other threads now blocked? What if two threads are waiting? Which thread gets the input?
  - What if one thread closes a file while another is still reading from it?
  - And so on…

A few more details on User-Level Threads

- Thread table contains entry for each thread:
  - Program ctr, stack ptr, state, thread context, etc.
- Run-time system (like a small operating system) manages threads within the process
  - Implements communication, synchronization among threads
  - Schedules and dispatches threads (thread switching)
- Advantages over process-based design, or even kernel-supported threads?

no kernel support required.
interaction/sync faster because all you do is function calls.
thread creation.

Problems with User-Level Threads

- Blocking system calls
  - One blocked thread will block all threads in process
  - Possible solution: Wrap such calls with code to check in advance whether call will block. How? Select system call select()
- Other blocking situation?

  I/O exceptions, page fault
- How to implement time slicing?

  sigalarm
- Problems arise for I/O bound applications
  - System call overhead subsumes cost of switching threads
  - Why not just support threads in kernel…

Kernel-supported threads

- Kernel maintains context of each thread within a process
- Thread creation, suspension, synchronization, are implemented with system calls to the kernel.
- Threads scheduled independently!
- Solves the blocking problem. Disadvantages? slower creation time. interaction
- Issues difficult with either type thread: some library routines may not be reentrant, handling per process entities such as signals, open files…
- ROOT of the problem???

  processes exist before multiple threads.
  threads added after the fact.

Threads in Linux

- Like any other operating system, Linux supports user-level threads
- With the clone() system call, also supports kernel-level threads
  - Creates child process (with own task_struct) in same address space of parent
  - Used to partially support POSIX threads
- In early 2000s, Native POSIX Thread Library (NPTL) developed to fully support threads
- POSIX threads, or ***Pthreads***

Example: POSIX Threads

- To support thread portability and uniformity, at least among Unix versions, IEEE defined the POSIX thread standard (*Pthreads*)
- Standardizes 60 function calls and various thread attributes and semantics
- Examples:

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Simple Pthreads Program

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
. . .
                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```


Other Interfaces

- Threads are supported in various programming languages and libraries, e.g.,
  - C++11, std::thread
  - Java Thread Object
  - Boost C++ Library, boost::thread
- In most cases, the programming primitives simply "wrap" a lower level implementation such as Pthreads
  - Advantage? Schedulable
- In CSE 410, we'll focus on Pthreads
  - Differences are largely just syntax issues…

Last comment: Multithreaded Kernel

- Supporting kernel reentrancy and even kernel-level threads on a uniprocessor is one thing…
  - Don't have to worry about preemption by other threads
  - Locks around data structures manipulated by interrupt handlers
  - Longer term locks (waiting for some resource) built into the logic of the kernel (sleep and wakeup)
- Implementing a multithreaded kernel is quite another.
- Supports concurrent execution of kernel code on multiple processors
- Approach: Lock everything!

Brief Introduction to Pthreads

(examples in C)

Pthreads

- Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls.
- This model is popular – nowadays practically all major thread libraries on Unix-like systems are pthreads-compatible
- In Linux, pthreads are realized as kernel-level threads.
  - Implications?

Process vs. Threads

- Threads share:
  - process instructions (text)
  - global data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - user and group id
- Unique to thread
  - thread ID
  - register context
  - stack
  - signal mask
  - priority, return value

Compiling pthread-based programs

- Include `pthread.h` in the main file
- Compile program with `-lpthread or -pthread`
  - `gcc -o test test.c -lpthread`
  - `g++ -pthread test.cpp`

Thread-related calls

- Types: `pthread_t` – type of a thread
- Some calls:

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

As with other primitives, always check return values on pthread functions

Thread features

- No explicit parent/child model, except main thread holds process info
- To enable other threads to continue execution after main thread is finished, need to call `pthread_exit` in main, don't just fall through
- Individual threads can call `pthread_exit` or just fall through (implicit call)
- `pthread_join` used to wait for another thread to exit
- Detached threads are those which cannot be joined (can also set this at thread creation)

Thread Creation and Termination

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

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     const char *message1 = "Thread 1";
     const char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Mutexes

- Used to
  - control access to shared resources
  - prevent race conditions
- Pthread mutexes work only among threads, not among processes
- Primitives:
  - `pthread_mutex_t mutex_variable = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_lock(&mutex_variable);`
  - `pthread_mutex_unlock(&mutex_variable);`
  - `pthread_mutex_trylock()` – tries but does not block…

Simple Mutex Example

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Join

- Wait for a thread to finish
- Example: wait for a collection of worker threads to finish their tasks
  - `pthread_join()` - wait for termination of another thread
  - `pthread_self()` - return identifier of current thread

Join Example

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

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id[j], NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */

   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

Thread Programming Problems

Logical execution is nondeterministic

- Race conditions (BTW, what is a race condition?)
  - Cannot predict how kernel schedules threads
  - Use mutexes and condition variables to help control it
- Code that is not thread-safe (e.g., reentrant)
  - Code was written to be linked to and used by a **process**
  - Called routines should not contain static or global variables unless they are protected by mutexes
  - Can protect the entire call with a mutex
- Mutex deadlock
  - Mutex is applied but not unlocked
  - Especially a problem when multiple mutexes are involved

Simple Example

- Grab one mutex
- If waiting for another, free first

```
...
pthread_mutex_lock(&mutex_1);
while ( pthread_mutex_trylock(&mutex_2) )  /* Test if already locked   */
{
   pthread_mutex_unlock(&mutex_1);  /* Free resource to avoid deadlock */
   ...
   /* stall here   */
   ...
   pthread_mutex_lock(&mutex_1);
}
count++;
pthread_mutex_unlock(&mutex_1);
pthread_mutex_unlock(&mutex_2);
...
```

Deadlock Example: Order is important

```
void *function1()
{
   ...
   pthread_mutex_lock(&lock1);          // Execution step 1
   pthread_mutex_lock(&lock2);          // Execution step 3 DEADLOCK!!!
   ...
   ...
   pthread_mutex_lock(&lock2);
   pthread_mutex_lock(&lock1);
   ...
}

void *function2()
{
   ...
   pthread_mutex_lock(&lock2);          // Execution step 2
   pthread_mutex_lock(&lock1);
   ...
   ...
   pthread_mutex_lock(&lock1);
   pthread_mutex_lock(&lock2);
   ...
}

main()
{
   ...
   pthread_create(&thread1, NULL, function1, NULL);
   pthread_create(&thread2, NULL, function2, NULL);
   ...
}
```


Summary

- What we have learned
  - Basic concept of multiple threads executing within one process
  - User-level vs. kernel-level threads
  - Contextual information needed for each thread
  - Speedup and Amdahl's Law
  - Semantics issues due to process model
  - Brief introduction to pthreads programming
  - Example usage of simple concurrency control primitives
- Next up: More complete treatment of concurrency
  ("meaty" topic applicable not only to threads!)