EECS 3221
Operating System Fundamentals
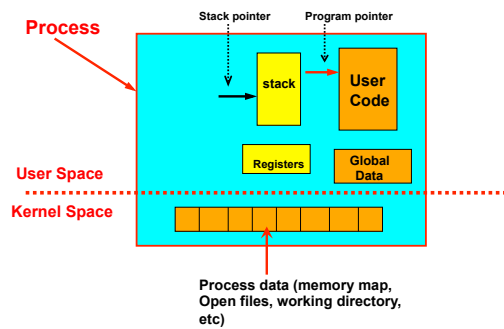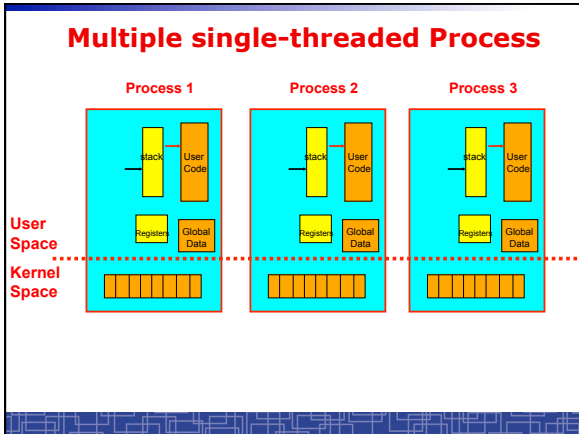
No. 3

Thread

Prof. Hui Jiang
*Dept of Electrical Engineering and Computer
Science, York University*
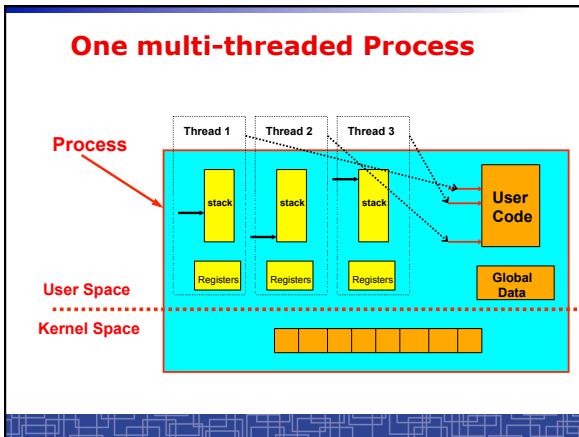

Thread Concept

- What is thread?

- Difference between a process and a thread

One single-threaded Process

Process

Stack pointer   Program pointer

stack   User Code

Registers   Global Data

User Space
Kernel Space

Process data (memory map,
Open files, working directory,
etc)

Multiple single-threaded Process

Process 1

Process 2

Process 3

stack | User Code

Registers | Global Data

User Space

Kernel Space

One multi-threaded Process

Thread 1 | Thread 2 | Thread 3

Process

stack | stack | stack

User Code

Registers | Registers | Registers
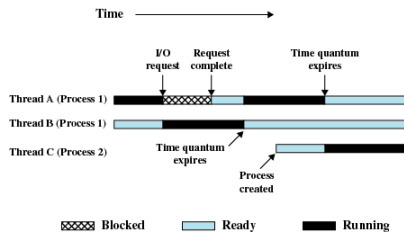
Global Data

User Space

Kernel Space

Process vs. Thread

- Traditional process contains a single stream of control.
  (one process can do one thing at a time)
- Multithreaded process: contains several different streams of control. Each stream is called a thread of this process.
  (multithreaded process can do multiple jobs simultaneously)
- A multi-threaded process contains several threads.
- All threads in a process share:
  – Code section & data section (global data & heap)
  – OS resources (memory map, open devices, accounting, etc.)
- Each thread includes:
  – A thread ID
  – A program counter (PC)
  – A register set
  – A stack & stack pointer

Comparison

- One single-threaded process:
  - can do one thing at a time

- Multiple single-threaded processes:
  - can do many things at the same time

- One multi-threaded process
  - Also can do many things at the same time

- Why multiple thread??
  - Multi-threaded process requires less OS resources (memory)
  - More efficient for OS to handle threads than processes

Multithreading

Time →

I/O request   Request complete   Time quantum expires

Thread A (Process 1)

Thread B (Process 1)

Thread C (Process 2)

Time quantum expires

Process created

▨▨▨ Blocked   ▭ Ready   ■ Running

Multithreading Example on a Uniprocessor

Benefits to use threads

- Threads occupy less memory than processes.

- Takes less time to create a new thread than a process.

- Less time to terminate a thread than a process.

- Less time to switch context between two threads within the same process.

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

Thread-safe or Reentrant code

- **To be thread safe, the program must be reentrant:**

    – **Program never modifies itself.**

    – **Each function calling keeps track of its own progress.**

    – **No use of static/global data.**

    – **No use of non-reentrant functions or routines.**

Non-reentrant C code

```
int delta;


int diff (int x, int y)
{

    delta = y - x;

    if (delta < 0) delta = -delta;

    return delta;

}
```

Reentrant C code

```
int diff (int x, int y)
{
    int delta;


    delta = y - x;

    if (delta < 0) delta = -delta;

    return delta;

}
```
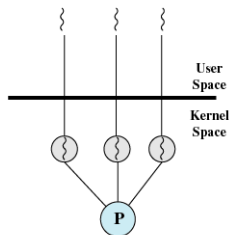
Kernel Threads

- Kernel threads are supported directly by OS kernel.
- The kernel performs thread creation, scheduling, and management in the kernel space.
- Slow to maintain (need system calls to kernel space).
- Each kernel thread can run totally independently:
  - One thread blocks, the kernel will schedule another thread to run.
  - Several kernel threads can run in parallel if many CPU's are available.
  - OS to support kernel thread:
    - Windows NT/2000/XP
    - Solaris 2
    - Linux

Directly Use Kernel Threads

- For each user task, make system call to create a kernel thread.

(b) Pure kernel-level

Example of Kernel Thread: Linux Thread

- Linux kernel support kernel threads, system call *clone()*.
- *fork()* creates a new process
  - Create a new memory space for new process
  - Copy from the address space of the calling process
- *clone()* simulates *fork()*, but
  - It does not create new memory space.
  - The new process shares the same address space of the original process.
  - → two processes sharing the same memory space.
    (something like thread)

Linux Thread

- **Linux use `clone()` to create kernel threads.**

```
#include <sched.h>
 int clone(int (*fn)(void *), void
 *child_stack, int flags, void *arg);
```

fn: starting function

child_stack: stack memory space for child thread.

flags: what to share.

   for thread creation:

   flags = CLONE_FS | CLOSE_VM | CLONE_SIGHAND |
         CLONE_FILES
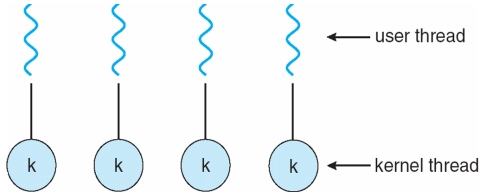
arg: arguments to pass.

User Thread

- **User thread: supported above the kernel and implemented by a thread library in user space.**
  - **The library supports thread creation, scheduling, management in user space.**
  - **User threads are fast to create and manage (no need to make a system call to trap to the kernel).**
  - **User threads for better compatibility across OS platforms.**
  - **User threads save kernel resources.**

- **Problems with user threads:**
  - **The kernel is not aware of the existence of users threads.**
  - **User thread must be mapped to the kernel to execute in CPU.**

- **Examples: POSIX Threads (Pthreads), Java Threads, Win32 Threads, Solaris UI-threads**

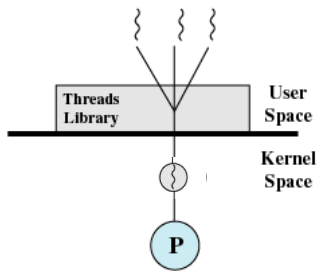## Three Models for User Thread

- **One-to-One mapping**

- **Many-to-One Mapping**

- **Many-to-Many Mapping**

One-to-One Mapping
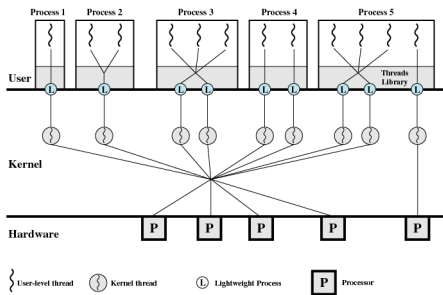

user thread

k k k k ← kernel thread

Many-to-One Mapping

Threads Library

User Space

Kernel Space

P

Combined Model: many-to-many mapping

user thread

k k k ← kernel thread

Threads Library

User Space

Kernel Space

P P

Solaris Threads

	Process 1	Process 2	Process 3	Process 4	Process 5

User — Threads Library

Kernel

Hardware — P P P P P

{ User-level thread   (S) Kernel thread   (L) Lightweight Process   [P] Processor

---
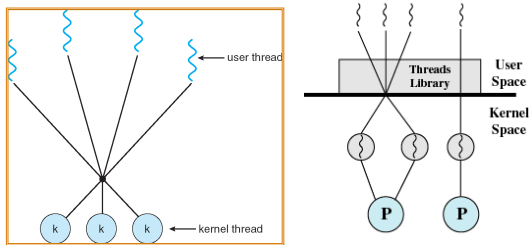
Thread data structure in Solaris

process id
memory map
priority
list of open files

LWP ID
Registers
priority
Kernel Stack
...

LWP₁   LWP₂   LWP₃   ...

Solaris process
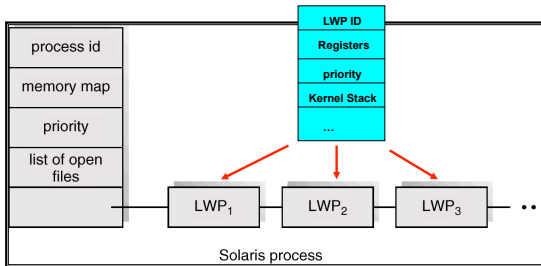
Threading Issues

- *fork()* and *exec()* **implementation**
  - **One thread calls** *exec()*, **it will replace the entire process.**
  - **One thread in a process call** *fork()*, **it duplicates all threads in the process or just one calling thread.**

- **Thread cancellation: terminating a thread before it finishes.**
  - **Asynchronous cancellation**
  - **Deferred cancellation**

- **Unix Signal Handling**
  - **Deliver the signal to the thread to which the signal applies.**
  - **Deliver the signal to every thread in the process**
  - **Deliver the signal to certain threads in the process**
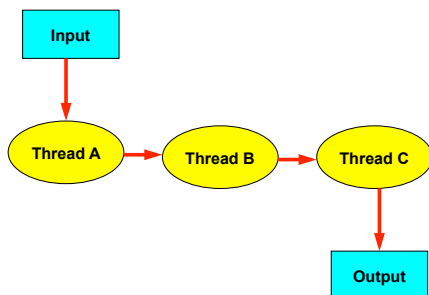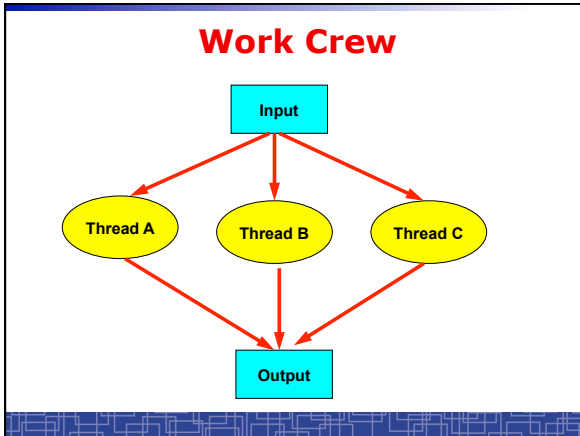 - **Assign a specific thread to receive all signals for the process**

Thread Pools

- Create a number of threads at process start-up, place them into a pool, where they sit and wait for work.
- When the process receives a request, it awakens a thread from the pool, and serves the request immediately.
- Once the thread completes, it returns to the pool.
- If the pool contains no available thread, the process waits until one becomes free.
- Benefits of thread pools:
  - Faster to service a request.
  - Thread pool limits the total number of threads in system (no overload).

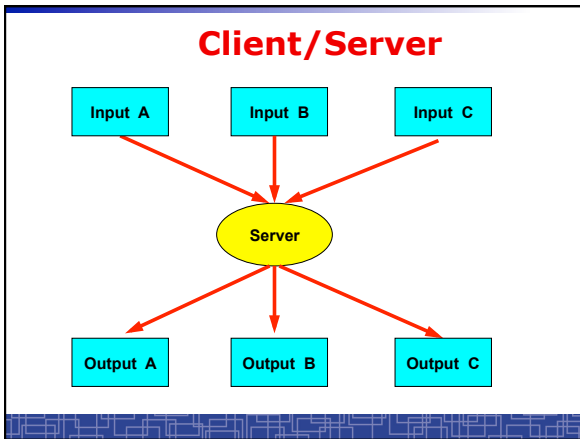## Three Models to use Threads

- Pipeline
  - Assembly line: each thread repeatedly performs the same operation on a sequence of data sets, passing each result to another thread for next step.

- Work Crew
  - Each thread performs an operation on its own data independently, then combine all results to get the final.

- Client/Server
  - A client contacts with an independent server for each job.

Pipeline

Work Crew

```
                    ┌────────┐
                    │ Input  │
                    └────────┘
         ┌──────────────┼──────────────┐
         ▼              ▼              ▼
   ┌──────────┐   ┌──────────┐   ┌──────────┐
   │ Thread A │   │ Thread B │   │ Thread C │
   └──────────┘   └──────────┘   └──────────┘
         └──────────────┼──────────────┘
                        ▼
                   ┌────────┐
                   │ Output │
                   └────────┘
```

Client/Server

```
┌──────────┐   ┌──────────┐   ┌──────────┐
│ Input A  │   │ Input B  │   │ Input C  │
└──────────┘   └──────────┘   └──────────┘
         \            │            /
          \           ▼           /
           ▼      ┌────────┐     ▼
                  │ Server │
                  └────────┘
          /           │           \
         ▼            ▼            ▼
┌──────────┐   ┌──────────┐   ┌──────────┐
│ Output A │   │ Output B │   │ Output C │
└──────────┘   └──────────┘   └──────────┘
```
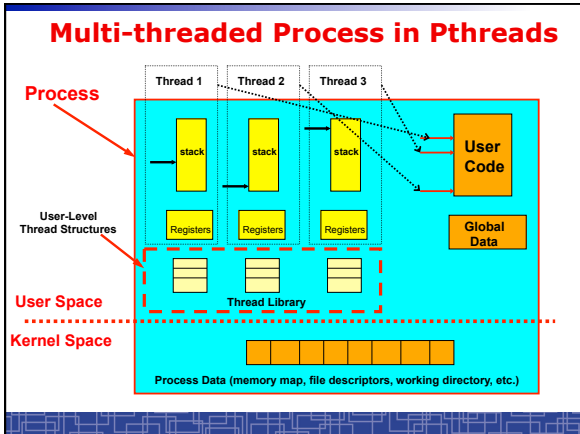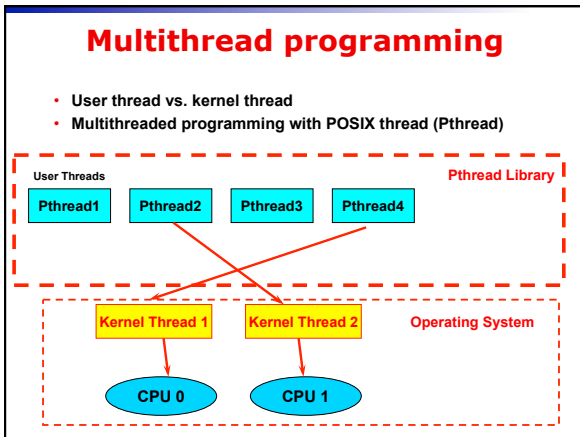
User Threads: Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

- API specifies behavior of the thread library, implementation is up to development of the library.

- Common in UNIX operating systems (Solaris, Linux, Mac OS X).

Multi-threaded Process in Pthreads

Process

Thread 1 Thread 2 Thread 3

stack stack stack

User Code

Registers Registers Registers

Global Data

User-Level Thread Structures

Thread Library

User Space

Kernel Space

Process Data (memory map, file descriptors, working directory, etc.)

Multithread programming

- **User thread vs. kernel thread**
- **Multithreaded programming with POSIX thread (Pthread)**

User Threads

Pthread Library

Pthread1 Pthread2 Pthread3 Pthread4

Kernel Thread 1 Kernel Thread 2 **Operating System**

CPU 0 CPU 1

POSIX Thread (1)

- **Thread creation and termination:**

```
#include <pthread.h>

pthread_create(pthread_t *thread, const pthread_attr_t
     *attr, void *(*start) (void *), void *argv) ;


          pthread_exit(void *value_ptr) ;
```

POSIX thread(2)

- Wait for another thread to terminate

pthread_join(pthread_t thread, void **value_ptr) ;

- Cancellation

pthread_cancel(pthread_t thread) ;

- Others

pthread_self(void) ;

pthread_detach(pthread_t thread) ;

pthread_attr_init(pthread_attr_t *attr) ;

Example 1: thread.c

- Example: **thread.c** (How to use pthread)

- Two threads:
  - *main*() thread
  - *runner*() thread

Example 2: alarm.c

- Example 1: **alarm.c** (no process/thread)

- Example 2: **alarm_fork.c** (multiple process)

- Example 3: **alarm_thread.c** (multiple thread)