Lecture Topics

- Today: Threads
  (Stallings, chapter 4.1-4.3, 4.6)

- Next: Concurrency
  (Stallings, chapter 5.1-5.4, 5.7)

Announcements

- Make tutorial
- Self-Study Exercise #4
- Project #2 (due 9/21)
- Project #3 (due 9/28)
UNIX Process Management

- Two categories of processes
  - System processes
    - run in kernel mode for housekeeping functions
      (memory allocation, process swapping, etc)
  - User processes
    - run in user mode for user programs
    - run in kernel mode for system calls, traps, interrupts

UNIX System Processes

- Process 0 is created at boot time and becomes the "swapper" process after forking process 1 (the "systemd" process) and process 2 (the "kthread" process)

- The "kthread" process is the ancestor of all kernel processes

- The "systemd" process is the ancestor of all user processes
UNIX Process Creation

- Every process (except process 0) is created by the fork() system call
  - fork() allocates entry in process table and assigns a unique PID to the child process
  - child gets a copy of the process image of the parent; both child and parent are executing the same code after fork()
  - fork() returns the PID of the child process to the parent process and returns 0 to the child process

```c
int pid = fork();

if (pid < 0)
{
    // Error -- fork failed
}
else if (pid > 0)
{
    // Parent process (pid contains child's PID)
}
else
{
    // Child process
}
```
UNIX Process Creation

- When the child process terminates, the parent process is notified.

- Typically, the parent process uses the \texttt{wait()} system call to voluntarily block until the child process terminates.

- If the parent process terminates before the child process, the child becomes an orphan process and is adopted by process 1.

UNIX Process Specialization

- The \texttt{fork()} system call creates a duplicate copy of the process image of the parent; both child and parent are executing the same code after \texttt{fork()}.

- The \texttt{exec()} family of system calls is used to replace the process image of one process (usually the child) with a different process image (a different program).
UNIX Process Specialization

```c
int pid = fork();

if (pid < 0)
{
    // Error -- fork failed
}
else if (pid > 0)
{
    // Parent process (pid contains child's PID)
}
else
{
    flag = execvp( "ls", args );
}
```

Main Process Characteristics

- **Resource ownership:**
  - virtual address space to hold the process image
  - control of some resources (files, I/O devices...)

- **Execution:**
  - path (trace) through object code
  - execution interleaved with other processes
  - execution context saved in PCB when the process is interrupted
Main Process Characteristics

- In modern operating systems, these two characteristics are often treated independently

- The unit of dispatching is usually referred to as a thread or (lightweight process)

- The unit of resource ownership is usually referred to as a process (or task)

Multithreading vs Single Threading

- Single threading: the OS does not recognize the concept of threads

- MS-DOS supports a single thread

- Traditional UNIX supports multiple user processes but only supports one thread per process

- Linux, Windows, Solaris, Mach, and OS/2 support multiple threads
Processes and Threads

Process

- A virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources
Thread

- An execution state (running, ready, etc.)
- Saved thread context when not running
- An execution stack
- Static storage for local variables
- Access to memory and resources of its process (shared by all threads)

Thread

- Viewpoint: thread is an independent program counter (and other context) within a process
- All threads within a process share the process resources
  - when one thread alters a (non-private) memory item, all other threads (of the process) see it
  - a file opened by one thread is available to others
Benefits of Threads

- Less effort to create a new thread than a new process
- Less effort to terminate a thread
- Less effort to switch 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
Benefits of Threads

- Consider an application that consists of several independent parts that do not need to run in sequence
  - Each part can be implemented as a thread
  - Whenever one thread is blocked waiting for I/O, execution could switch to another thread of the same application (instead of switching to another process)

Example Applications

- Web server
  - must handle several requests over a short period
  - more efficient to create (and destroy) a single thread for each request

- Menu-driven program
  - thread to display menu, read user input
  - thread(s) to execute user commands
Linux Processes and Threads

- There is no distinction between processes and threads in Linux: the term task is used to refer to both.

- One task can create another task by cloning itself (clone() is a more flexible version of fork() – can specify degree of resource sharing).

---

Linux Task States

[Diagram showing task states: Ready, Executing, Stopped, Uninterruptible, and Zombie, with transitions indicated by arrows for signal, event, scheduling, termination, etc.]
Linux Task States

- Ready – ready to use the CPU
- Running – using the CPU
- Uninterruptible – waiting on hardware
- Interruptible – waiting for resource or signal
- Stopped – waiting for another process to restart it (for debugging)
- Zombie – terminated, waiting for system to get info from PCB

Examples

- threads1.c
- threads2.c
- threads3.c
- threads4.c
- threads5.c
- threads6.c
Not Covered at Lecture

- The following slides were not covered at lecture on 9/21 (no time)
- They are included here to support the readings in Chapter 4 of Stallings

Evolution of Thread Management

- Initially, threads were implemented in a library that did not rely on kernel support
- Over time, support of threads was added to the kernel, leading to several methods for handling threads
User-Level Threads

- Thread management is done by the application
- The kernel is not aware of the existence of threads

Threads Library

- Contains code for:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts
Kernel Activity for ULTs

- The kernel is not aware of thread activity but it is still manages process activity

- When a thread makes a system call, the whole process will be blocked

Advantages of ULTs

- Thread switching does not involve the kernel: no mode switch or process switch

- Thread scheduling can be application specific: programmer chooses best scheduling algorithm

- ULTs can run on any OS: just need threads library
Inconveniences of ULTs

- Most system calls are blocking and the kernel blocks processes: all threads within the process will be blocked
- The kernel can only assign processes to processors: two threads within the same process cannot run simultaneously on two different processors

Kernel Level Threads

- All thread management is done by the kernel
- No thread library but an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling on a per thread basis
Kernel-Level Threads

Advantages of KLTs

- The kernel can simultaneously schedule many threads of the same process on many processors
- Blocking is done on a thread level: if one thread is blocked, other threads can still continue
- Kernel routines can be multithreaded
Inconveniences of KLTs

- Additional overhead: switching between threads within the same process involves the kernel

Combined Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- Multiple ULTs are mapped onto smaller or equal number of KLTs
- The programmer may adjust the number of KLTs
- Combines the best of both approaches
Summary

- A process can contain one or more threads
- Multiple threads wander through the same program concurrently, maybe simultaneously!
User-Level vs. Kernel-Level Support

- **ULT**: kernel just sees process
- **KLT**: kernel maintains info on each thread

![Diagram showing user-level vs. kernel-level support]