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 9/20)
- Project #3 (due 9/27)
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 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 fork() system call creates a duplicate copy of the process image of the parent; both child and parent are executing the same code after fork().

- The 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
ing 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
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
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