CSCE 313 Introduction to Computer Systems

Instructor: Dr. Guofei Gu

http://courses.cse.tamu.edu/guofei/csce313/

Programs, Processes, and Threads

- Programs and Processes
- Threads
Programs, Processes, and Threads

- Programs and Processes
- Threads

Processes Management

- What is a process?
- How to control processes.
- How to allocate the available resources to the execution of the processes (scheduling)
- How to coordinate processes among themselves (synchronization)
Processes and Process Control

- Q: What is a process?

- *Process* as execution of a *Program*

- We can *trace* the execution of a process

- Process as *minimal entity for resource allocation* (for example memory).

---

Simple Memory Layout of a Running Program

<table>
<thead>
<tr>
<th>high address</th>
<th>command-line arguments and environment variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>stack</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>heap</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>uninitialized static data</td>
</tr>
<tr>
<td></td>
<td>initialized static data</td>
</tr>
<tr>
<td></td>
<td>program text</td>
</tr>
</tbody>
</table>
# The Execution Trace of Processes

- Two processes and a dispatcher

<table>
<thead>
<tr>
<th>States of a Process</th>
</tr>
</thead>
</table>
- **User view:** A process is executing continuously
- **In reality:** Several processes compete for the CPU and other resources
- A process may be
  - **running:** it holds the CPU and is executing instructions
  - **Blocked(waiting):** it is waiting for some I/O event to occur
  - **ready:** it is waiting to get back on the CPU
Process Switch

- Mechanism of a process switch:

  - Preempt Process A and store all relevant information.
  - Load information about Process B and continue execution.

  - Preempt Process B and store all relevant information.
  - Load information about Process A and continue execution.

- The PCB contains all information specific to a process.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.
- Context of a process represented in the PCB.
  - Context-switch time is overhead; the system does no useful work while switching.
  - Time dependent on hardware support.
Process Control Block (PCB)

| process identification | process id  |
|                        | parent process id |
|                        | user id         |
|                        | etc…            |
| processor state information | register set condition codes |
| processor control information | processor status |
| process control information | process state |
|                             | scheduling information |
|                             | event (wait-for) |
|                             | memory-mgmt information |
|                             | owned resources (e.g., list of opened files) |

Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
Example for the Use of PCBs: Process Queues

Scheduler

- **Long-term scheduler** (or job scheduler)
  - selects which processes should be brought into the ready queue
  - controls degree of multiprogramming
  - must select a good process mix of I/O-bound and CPU-bound processes
- **Short-term scheduler** (or CPU scheduler)
  - selects which process should be executed next and allocates CPU
  - executes at least every 100ms, therefore must be very fast
- **Medium-term scheduler** (swapper)
  - in some Oss
  - sometimes good to temporarily remove processes from memory (suspended)
Suspended Processes

Process Creation

- **When?**
  - Submission of a batch job
  - User logs on
  - Create process to provide service such as printing
  - Spawned by existing processes

- **How?**
  - In UNIX: all processes created by `fork()` system call
Process Creation (Cont)

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes.
- Generally, process identified and managed via a **process identifier** (pid).
- Resource sharing:
  - Parent and children share all resources.
  - Children share subset of parent’s resources.
  - Parent and child share no resources.
- Execution:
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

---

Process Creation (Cont)

- Address space:
  - Child duplicate of parent.
  - Child has a program loaded into it.

- UNIX examples:
  - `fork` system call creates new process.
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program.
Process Creation

C Program Forking Separate Process

```c
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```
Program Example 1 (chain)

```c
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2){   /* check for valid number of command-line arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;

    fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

Program Example 2 (fan)

```c
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2){   /* check for valid number of command-line arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if ((childpid = fork()) <= 0)
            break;

    fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```
Some work for you

- How to create the following process tree?

A tree of processes on a typical Solaris
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process’ resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

Processes in UNIX

- **fork()**
- **created**
  - **not enough memory**
  - **ready swapped**
  - **swap in**
  - **wake up**
  - **sleep swapped**
- **enough memory**
  - **ready**
  - **swap out**
  - **wake up**
  - **sleep in memory**
- **preempted**
  - **preempt**
  - **interrupt**
  - **system call**
  - **user running**
- **kernel running**
  - **reschedule process**
  - **return**
- **return to user**
Programs, Processes, and Threads

- Programs and Processes
- Threads

Traditionally, processes interact very little:

- This is not true in modern systems: Some applications may want to have multiple, tightly-coupled processes.
A thread is a flow of control within a process

Problems with traditional (heavy-weight) processes

- Heavy-weight processes have separate address spaces:
  - Process creation is expensive
  - Process switch is expensive
  - Sharing memory areas among processes non-trivial
Threads

- Threads share address space:
  - Thread creation much simpler than process creation (no need to create and initialize address space, etc.)
  - Thread switch simple
  - Threads fully share the address space

- Convenience
  - Communication between threads

- Efficiency
  - Multiprogramming within a process
  - Multiprocessors

Benefits

- Responsiveness
  - Increase responsiveness for a multithreaded interactive application (e.g., web browser)

- Resource Sharing
  - Share memory and resources (e.g., code, data)

- Economy (creation and context-switch)
  - In Solaris, creating a process is about 30 times slower than creating a thread; context switching is 5 times slower

- Utilization of multiprocessor architecture (Scalability)
Multithreaded Server Architecture

Concurrent Execution on a Single-core System
Parallel Execution on a Multicore System

User-Level vs. Kernel-Level Threads

- **User-level**: kernel not aware of threads
- **Kernel-level**: all thread-management done in kernel
User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
Multithreading Models

- Many-to-One
  - Many user-level threads mapped to single kernel thread
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads
- One-to-One
  - Each user-level thread maps to kernel thread
  - Examples
    - Windows NT/XP/2000
    - Linux
    - Solaris 9 and later
- Many-to-Many

Potential Problems with Threads (many-to-one)

- General: Several threads run in the same address space:
  - Protection must be explicitly programmed (by appropriate thread synchronization)
  - Effects of misbehaving threads limited to task
- User-level threads: Some problems at the interface to the kernel: With a single-threaded kernel, as system call blocks the entire task.
Protection of kernel data structures is trivial, since only one process is allowed to be in the kernel at any time.

Concurrency?

Special protection mechanism is needed for shared data structures in kernel.

Concurrency?

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Most efficient, also most complex
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-Level Model: Threads in Solaris 2.x
Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread pools
- Thread-specific data

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Two versions of fork
    - Duplicates all threads
    - Duplicates only the calling thread

- Exec() will replace all the process —including all threads
Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Operating System Examples

- Windows XP Threads
- Linux Thread
Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)
Linux Threads

- Linux typically refers to threads as *tasks*
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>