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**

```
| high address | command-line arguments and environment variables |
|              | stack                                             |
|              | heap                                              |
| low address  | uninitialized static data                         |
|              | initialized static data                           |
|              | program text                                      |
```
The Execution Trace of Processes

- Two processes and a dispatcher

<table>
<thead>
<tr>
<th>program B</th>
<th>Traces of processes A and B</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>β</td>
</tr>
<tr>
<td>α+1</td>
<td>β+1</td>
</tr>
<tr>
<td>α+2</td>
<td>β+2</td>
</tr>
<tr>
<td>α+3</td>
<td>β+3</td>
</tr>
<tr>
<td>α+4</td>
<td>β+4</td>
</tr>
<tr>
<td>α+5</td>
<td>β+5</td>
</tr>
<tr>
<td>α+6</td>
<td>β+6</td>
</tr>
<tr>
<td>α+7</td>
<td>β+7</td>
</tr>
<tr>
<td>α+8</td>
<td>β+8</td>
</tr>
<tr>
<td>α+9</td>
<td>β+9</td>
</tr>
<tr>
<td>α+10</td>
<td>β+10</td>
</tr>
<tr>
<td>α+11</td>
<td>β+11</td>
</tr>
</tbody>
</table>

Trace of dispatcher

- program A
- dispatcher

<table>
<thead>
<tr>
<th>δ</th>
<th>β+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ+1</td>
<td>β+2</td>
</tr>
<tr>
<td>δ+2</td>
<td>β+3</td>
</tr>
<tr>
<td>δ+3</td>
<td>β+4</td>
</tr>
</tbody>
</table>

States of a Process

- **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

Create (new) − Ready − Running − Exit (terminated)

Preempt − Dispatch − I/O or event wait
Process Switch

- Mechanism of a process switch:

Process A

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

Process B

- 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 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**

- **ready**
- **running**
- **waiting**
  - ready queue
  - executing process
  - disk 1
  - disk 2
  - I/O device queues
  - serial I/O

**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
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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?

```
  0
 /|
/ | \\
2a 3a
 |
 |
|
2b
 |
 |   \
3b 3c 3d
```

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
- swap out
- awake
- sleep
- swapped
- sleep in memory
- preempted
- preemption
- enough memory
- ready
- reschedule
- system call
- kernel running
- swap in
- swap out
- awake
- sleep
- exit
- user running
- return to user
- return
- zombie
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.

- 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

(1) request

client ➔ server ➔ thread

(2) create new thread to service the request

(3) resume listening for additional client requests

Concurrent Execution on a Single-core System

single core: T₁, T₂, T₃, T₄, T₁, T₂, T₃, T₄, T₁, ...

time
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.
Singlethreaded vs. Multithreaded Kernel

- 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>