Processes and Threads

[SGG7] Chapters 3 and 4

Process Concept

- Process = a program in execution
- A process includes:
  - text section (loaded program)
  - program counter
  - stack (+ heap)
  - data section (global variables)

Process State

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a process
  - terminated: The process has finished execution

Process Control Block (PCB)

Information associated with each process
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

CPU Switch From Process to 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
- Context-switch time is **overhead**
  - the system does no useful work while switching
  - time depends on hardware support

**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

**Schedulers**
- **Long-term scheduler** (or job scheduler)
  - for batch systems – new jobs for execution queued on disk
  - selects which processes should be brought into the ready queue, and loads them into memory for execution
  - controls the degree of multiprogramming
  - invoked very infrequently (seconds, minutes)
  - No long-term scheduler on UNIX and Windows; instead **swapping**, controlled by medium-term scheduler
- **Short-term scheduler** (or CPU scheduler)
  - selects which ready process should be executed next
  - invoked very frequently (milliseconds)
  ⇒ must be fast

**CPU-bound vs I/O-bound processes**
- **I/O-bound process**
  - spends more time doing I/O than computations
  - many short CPU bursts
- **CPU-bound process**
  - spends more time doing computations;
  - few very long CPU bursts
- **Long-term (or medium-term) scheduler** should aim at a good **process mix.**

**Scheduling**
- **Non-preemptive scheduling:**
  - process keeps CPU until it terminates or voluntarily releases it (sleep) – step back into ready queue
- **Preemptive scheduling:**
  - OS puts process from CPU back into ready queue after a certain time quantum has passed

**More about this in the lecture on Scheduling**
**Process Creation**

- **Parent process** creates **children processes**, which, in turn create other processes, forming a **tree of processes**.
- **Resource sharing** variants:
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- **Execution** variants:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address space** variants:
  - Child is a duplicate of parent
  - Child has a program loaded into it

**Example: Process Creation in UNIX**

```c
int main()
{
    Pid_t ret;
    /* fork another process: */
    ret = fork();
    if (ret < 0)  {  /* error occurred */
        fprintf ( stderr, "Fork Failed" );
        exit(-1);
    }
    else if (ret == 0)  {
        /* I am child process */
        execlp ( "/bin/ls", "ls", NULL );
    }
    else {  // I am the parent process
        /* wait for child to complete: */
        wait(NULL);
        printf ("Child complete" );
        exit(0);
    }
}
```

**A typical tree of processes in Solaris**

**Process Termination**

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Process returns status value to its parent (used in **wait**)
  - OS de-allocates process’s resources
- 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 OS do not allow child to continue after parent terminates
    - All children terminated - cascading termination

**Cooperating Processes**

- **Independent** process
  - cannot affect or be affected by execution of another process
- **Cooperating** process
  - can affect or be affected by execution of another process
- **Advantages of process cooperation**:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- **Inter-Process Communication (IPC)**
  - shared memory
  - message passing
  - signals

**IPC Models – Realization by OS**

- **IPC via Message Passing**
- **IPC via Shared Memory**
Example: POSIX Shared Memory API

- `#include <sys/shm.h>`
- `#include <sys/stat.h>`
- Let OS create a shared memory segment (system call):
  - `int segment_id = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- Attach the segment to the executing process (system call):
  - `void *shmemptr = shmat (segment_id, NULL, 0);`
- Now access it:
  - `strcpy ((char *)shmemptr, "Hello world"); // Example: copy a string into it`
- Detach it from executing process when no longer accessed:
  - `shmdt (shmemptr);`
- Let OS delete it when no longer used:
  - `shmctl (segment_id, IPC_RMID, NULL);`

Example for IPC: Producer-Consumer Problem

- **Producer-Consumer paradigm** for cooperating processes:
  - *producer* process produces data items that are consumed by a *consumer* process
- **Realization with shared memory:**
  - *unbounded-buffer*
    - places no practical limit on the size of the buffer
    - Consumer must wait when buffer is empty
  - *bounded-buffer*
    - assumes that there is a fixed buffer size
    - Producer must also wait when buffer is full

Bounded-Buffer – Shared-Memory Solution

- **Shared buffer:**
  ```c
  #define BUFFER_SIZE 10
  typedef struct {
      ...
  } item;
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

| Buffer empty when in == out |
| Buffer full when ((in+1) % BUFFER_SIZE) == out |
| Can hold at most BUFFER_SIZE – 1 elements |

IPC with Message Passing

- **Message system**
  - processes communicate with each other without resorting to shared variables
  - provides two basic operations:
    - `send(receiverPID, message)`
    - `receive(senderPID, message)`
- In order to communicate, two processes
  - establish a communication link between them
  - exchange messages via send/receive

Client-Server Communication

- **Message passing variant for client-server systems**
  - **Sockets**
    - Endpoint for IPC between clients and servers
    - addressed by (IP address, port number) instead of PID
  - **Remote Procedure Calls**
    - Client calls function of (maybe remote) server process by sending a RPC request to a server socket address
    - Server listens on socket port for incoming RPC requests
  - In Java: Remote Method Invocation (RMI)

To be continued in lecture on synchronization...
Threads – Overview

Thread Concept
Multithreading Models
Threading Issues
Thread libraries
- Pthreads [SGG7] 4.3.1
- Win32 Threads [SGG7] 4.3.2
- Java Threads [SGG7] 4.3.3
OS thread implementations
- Windows XP Threads [SGG7] 4.5.1
- Linux Threads [SGG7] 4.5.2

Single- and Multithreaded Processes

Benefits of Multithreading
Responsiveness
- Interactive application can continue even when part of it is blocked
Resource Sharing
- Threads of a process share its memory by default
Economy
- Light-weight
- Creation, management, context switching for threads is much faster than for processes
- E.g. Solaris: creation 30x, switching 5x faster
Utilization of Multiprocessor Architectures

User Threads (User-Level Threads)
Thread management (scheduling, dispatch) done by user-level threads library (linked with the application), without kernel support.
Kernel views all user threads of a multithreaded process as a single thread of control.
- process dispatched as a unit
- user control of scheduling algorithm; less overhead
- user threads do not scale well to multiprocessor systems
Three primary user-level thread libraries:
- Win32 threads
- Java threads
- POSIX Pthreads (API / standard, not implementation – may be provided as either user- or kernel-level library)

Kernel Threads (Kernel-Level Threads)
Managed by the OS kernel (Kernel-specific thread API)
Each kernel thread services (executes) one or several user threads
- Flexible: OS can dispatch ready threads of a multithreaded process even if some other thread is blocked.
- Kernel invocation overhead at scheduling/synchronization; less portable
Supported in
- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

Multithreading Models
Relationship user threads – kernel threads:
- Many-to-One (M:1)
- One-to-One (1:1)
- Many-to-Many (M:N)
Variations:
- Two-Level Model
- Light-Weight Processes [SGG7] 4.4.6
### Many-to-One
- Many user-level threads mapped to single kernel thread
- Low overhead
- Not scalable to multiprocessors
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

### One-to-One
- Each user-level thread maps to one kernel thread
- More concurrency; scalable to multiprocessors
- Overhead of creating a kernel thread for each user thread
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

### Many-to-Many Model
- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Solaris 8 and earlier

### Two-level Model
- Similar to Many-to-Many, except that it allows a user thread to be bound to a kernel thread
- Examples:
  - Solaris 8 and earlier
  - IRIX
  - HP-UX
  - Tru64 UNIX

### Threading Issues
- Semantics of `fork()` and `exec()` system calls
- Thread cancellation
- Signal handling
  - [SGG7] 4.4.3
- Thread pools
- Thread specific data
  - [SGG7] 4.4.5
- Scheduler activations
  - [SGG7] 4.4.6

### Semantics of `fork()` and `exec()`
- Does `fork()` duplicate only the calling thread or all threads?
- UNIX supports both variants (2 versions of `fork()`).
- If `exec()` used after `fork()` to replace process – no need to duplicate 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
- Mechanism: e.g., UNIX signal handling [SGG7] 4.4.3

Thread Pools

- For a multithreaded process: Create a number of threads in a pool where they await work
- Advantages:
  - Faster to service a request with an existing thread than to create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Win32 API
- OpenMP

User-level thread library example: Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior, not implementation, of the thread library
- C interface, e.g.
  ```c
  int pthread_create ( pthread_t *thread, const pthread_attr_t *attr, void (*start_routine, void *), void *arg);
  ```
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- [SGG7] 4.3.1
- In High-Performance Computing domain replaced by OpenMP (www.openmp.org)
- TDDC78

OS Thread Implementation Example: Linux Threads

- Linux does not distinguish between processes and threads.
  - Linux refers to them as tasks rather than threads
  - Degree of sharing can be controlled flexibly
- Common flags:
  ```
<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE_SYS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLOSE_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>
  ```
- Creation through clone() system call
  - allows a child task to share the address space etc. of the parent task (process)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java thread states:
  ```
  new => init => run => blocked => run => join => death
  ```
  [SGG7] 4.3.3

What have we learned?

- Processes versus Threads
- Process control block
- Context switch
- Ready queue and other queues used for scheduling
- Long-Mid-term versus Short-term scheduler
- Process creation and termination
- Process tree
- Inter-Process Communication
- Basics of Synchronization
- User threads versus Kernel threads
- Threading models: M:1, 1:M, N:M, two-level
- Examples: Pthreads API, Linux tasks