Processes and Threads

Giuseppe Anastasi
g.anastasi@iet.unipi.it
Pervasive Computing & Networking Lab. (PerLab)
Dept. of Information Engineering, University of Pisa

Based on original slides by Silberschatz, Galvin and Gagne

Overview

- Processes
- Threads
- CPU Scheduling

Overview

- **Processes**
- Threads
- CPU Scheduling

Process Concept

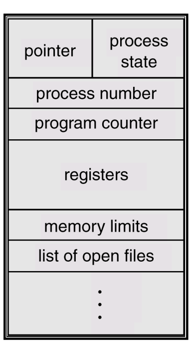
- Process – a program in execution;
  - Program is a passive entity (file on disk storage)
  - Process is an active entity
  - More processes can refer to the same program

Two instances of the same program (e.g., MS Word) have the same code section but, in general, different current activities

Process Concept (cont'd)

- A process includes
  - Code section
  - Current activity
- Current activity is defined by
  - Program Counter (IP Register)
  - CPU Registers
  - Stack
  - Data Section (global variables)
  - …

Process Control Block (PCB)



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

Process Creation

- Processes need to be created
  - Processes are created by other processes
  - System call `create_process`
- *Parent* process create *children* processes
  - which, in turn create other processes, forming a tree of processes.
- 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
  - Each process is identified by the *process identifier*
  - **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 in UNIX

```
# include <iostream.h>
void main(int argc, char* argv[]) {
    int pid;
    pid=fork(); /* genera un nuovo processo */
    if(pid<0) { /* errore */
          cout << “Errore nella creazione del processo” << "\n\n”;
          exit(-1);
    }
    else if(pid==0) { /* processo figlio */
          execlp(“/bin/ls”, “ls”, NULL);
    }
    else { /* processo genitore */
          wait(NULL);
          cout << “Il processo figlio ha terminato” << "\n\n”;
          exit(0);
    }
}
```


Process Termination

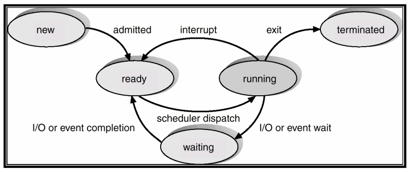
- Process terminates when executing the last statement
- The last statement is usually **exit**
  - Process’ resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

Process Evolution

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.

Diagram of Process State



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.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  - Terminates
  - Switches from running to waiting state
  - Switches from running to ready state
  - Switches from waiting to ready
- Scheduling under 1 and 2 is **nonpreemptive**
- All other scheduling is **preemptive**

Overview

- Processes
- **Threads**
- CPU Scheduling

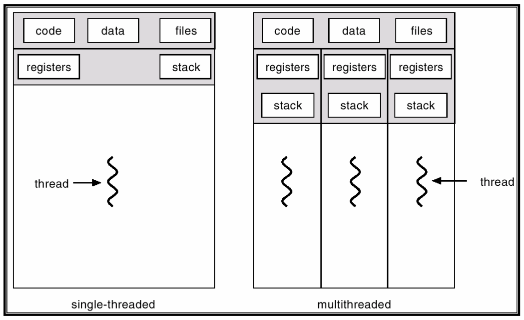
Process

- Resource ownership
  - A process is an entity with some allocated resources
    - Main memory
    - I/O devices
    - Files
    - …..
- Scheduling/execution
  - A process can be viewed as a sequence of states (execution path)
  - The execution path of a process may be interleaved with the execution paths of other process
  - The process is the entity than can be scheduled for execution

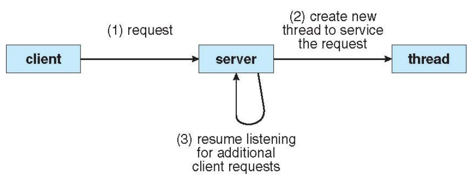
Processes and Threads

- In traditional operating systems the two concepts are not differentiated
- In modern operating systems
  - **Process**: unit of resource ownership
  - **Thread**: unit of scheduling
- Thread (Lightweight Process)
  - Threads belonging to the same process share the same resources (code, data, files, I/O devices, …)
  - Each thread has its own
    - Thread execution state (Running, Ready, …)
    - Context (Program Counter, Registers, Stack, …)

Single and Multithreaded Processes



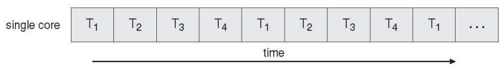
Multithreaded Server Architecture



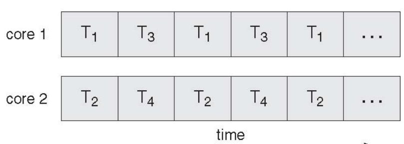
Benefits

- Responsiveness
  - An interactive application can continue its execution even if a part of it is blocked or is doing a very long operation
- Resource Sharing
  - Thread performing different activity within the same application can share resources
- Economy
  - Thread creation management is much easier than process creation and management
- Utilization of Multiple Processor Architectures
  - Different threads within the same application can be executed concurrently over different processors in MP systems

Execution on a Single-core System



Execution on a Multi-core System



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
  - Mac OS X
  - Linux
  - Solaris
  - Tru64 UNIX (Digital UNIX)

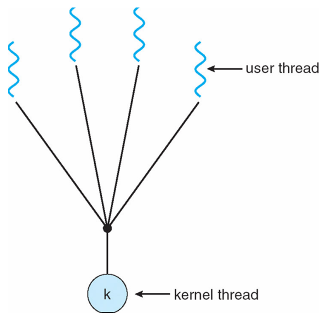
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

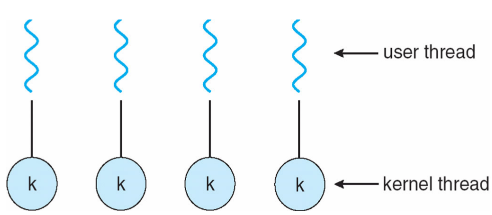
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

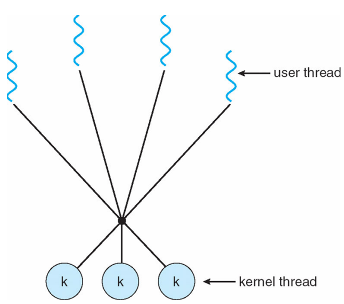
One-to-one Model



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
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

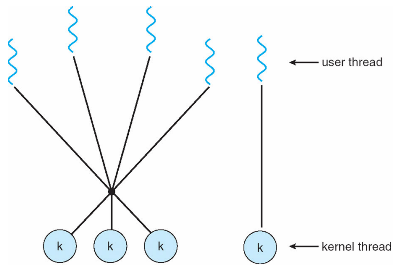
Many-to-Many Model



Two-level Model

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

Two-level Model



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

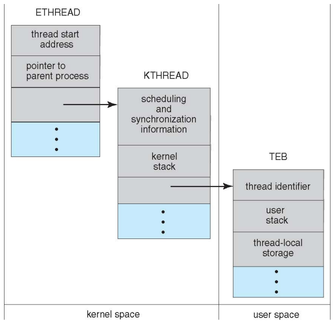
Operating System Examples

- 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 thread
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



Linux Threads

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

Linux Threads

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

Overview

- Processes
- Threads
- **CPU Scheduling**

CPU Scheduler

- Selects from among the ready processes and allocates the CPU to one of them.
- CPU scheduling decisions may take place in different situations
  - Non-preemptive scheduling
    - The running process terminates
    - The running process performs an I/O operation or waits for an event
  - Preemptive scheduling
    - The running process has exhausted its time slice
    - A process A transits from blocked to ready and is considered more important than process B that is currently running
    - …

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the scheduler; this involves:
  - Context Switch
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- *Dispatch latency*
  - time it takes for the dispatcher to stop one process and start another running.
  - should be minimized

Type of scheduling

- Batch Systems
  - Maximize the resource utilization
- Interactive Systems
  - Minimize response times
- Real-Time Systems
  - Meet Temporal Constraints

Objectives

- General
  - Fairness
  - Load Balancing (multi-processor systems)
- Batch Systems
  - CPU utilization (% of time the CPU is executing processes)
  - Throughput (# of processes executed per time unit)
  - Turnaround time (amount of time to execute a particular process)
- Interactive Systems
  - Response time
    - amount of time it takes from when a request was submitted until the first response is produced, **not** output
- Real-Time Systems
  - Temporal Constraints

Scheduling Algorithms

- Batch Systems
  - First-Come First-Served (FCFS)
  - Shortest Job First (SJF), Shortest Remaining Job First (SRJF)
  - Approximated SJF
- Interactive Systems
  - Round Robin (RR)
  - Priority-based
- Soft Real-Time Systems
  - Priority-based?

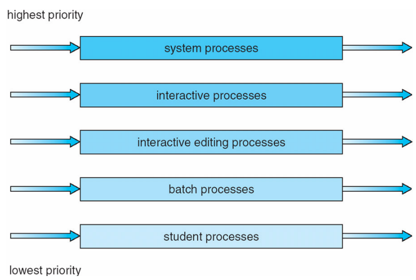
General-purpose systems

- General-purpose systems (e.g., PCs) typically manage different types of processes
  - Batch processes
  - Interactive processes
    - user commands with different latency requirements
  - Soft real-time processes
    - multimedia applications
- Which is the most appropriate scheduling in such a context?

Multi-level Ready Queue

- Ready queue is partitioned into separate queues
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling
    - Serve all from foreground then from background. Possibility of starvation.
  - Time slice
    - each queue gets a certain amount of CPU time (i.e., 80% to foreground in RR, 20% to background in FCFS)

Multilevel Queue Scheduling



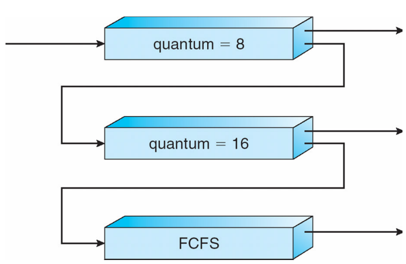
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



Operating System Examples

- Windows XP scheduling
- Linux scheduling

Windows XP Scheduling

- Thread scheduling based on
  - Priority
  - Preemption
  - Time slice
- A thread is execute until one of the following event occurs
  - The thread has terminated its execution
  - The thread has exhausted its assigned time slice
  - The has executed a blocking system call
  - A thread higher-priority thread has entered the ready queue

Kernel Priorities

- Kernel priority scheme: 32 priority levels
  - Real-time class (16-31)
  - Variable class (1-15)
  - Memory management thread (0)
- A different queue for each priority level
  - Queues are scanned from higher levels to lower levels
  - When no thread is found a special thread (idle thread) is executed

Win32 API priorities

- Priority classes
  - REALTIME_PRIORITY_CLASS → Real-time Class
  - HIGH_PRIORITY_CLASS → Variable Class
  - ABOVE_NORMAL_PRIORITY_CLASS → Variable Class
  - NORMAL_PRIORITY_CLASS → Variable Class
  - BELOW_NORMAL_PRIORITY_CLASS → Variable Class
  - IDLE_PRIORITY_CLASS → Variable Class
- Relative Priority
  - TIME_CRITICAL
  - HIGHEST
  - ABOVE_NORMAL
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE

Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Default Base Priority

Class Priority Management

- A thread is stopped as soon as its time slice is exhausted
- Variable Class
  - If a thread stops because time slice is exhausted, its priority level is decreased
  - If a thread exits a waiting operation, its priority level is increased
    - waiting for data from keyboard, mouse → significant increase
    - Waiting for disk operations → moderate increase
- Background/Foreground processes
  - The time slice of the foreground process is increased (typically by a factor 3)

Linux Scheduling

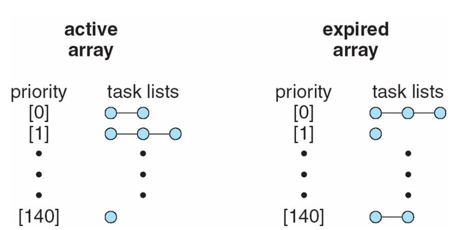
- Task scheduling based on
  - Priority levels
  - Preemption
  - Time slices
- Two priority ranges: real-time and time-sharing
  - **Real-time** range from 0 to 99
  - **Nice** range from 100 to 140
- The time-slice length depends on the priority level

Priorities and Time-slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

RunQueue

- The runqueue consists of two different arrays
  - Active array
  - Expired array



Priority Calculation

- Real time tasks have static priority
- Time-sharing tasks have dynamic priority
  - Based on nice value ± 5
  - ± 5 depends on how much the task is interactive
    - Tasks with low waiting times are assumed to be scarcely interactive
    - Tasks with large waiting times are assumed to be highly interactive
- Priority re-computation is carried out every time a task has exhausted its time slice

Questions?