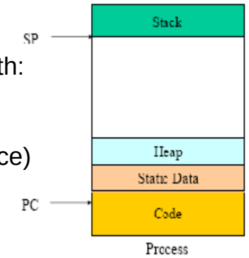
OS: Processes vs. Threads

CIT 595
Spring 2010

Process

- A process is a name given to a program instance that has been loaded into memory and managed by the operating system
- Process address space is generally organized into *code*, *data (static/global)*, *heap*, and *stack* segments
- Every process in execution works with:
  - Registers: PC, Working Registers
  - Call stack (Stack Pointer Reference)



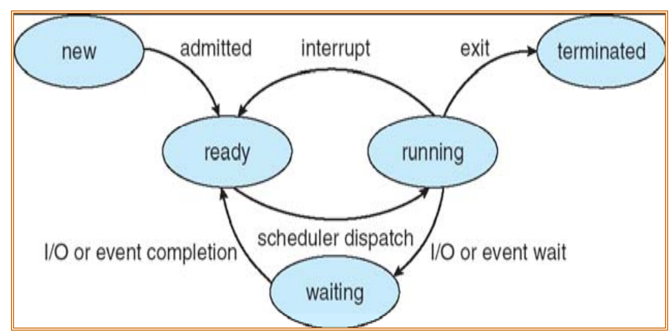
Process Activity

- As process executes over time it can be doing either of the activities or is in following states :

State/Activity	Description
new	Process is being created
running	Instructions are being executed on the processor
waiting/blocked	Process is waiting for some event to occur
ready	Process is waiting to use the processor
terminated	Process has finished execution

Note: state names vary across different OS

State Diagram of a Process Activity/State



Note: In uniprocessor system, only one process can be in running state while many processes can be in ready and waiting states

Process Creation

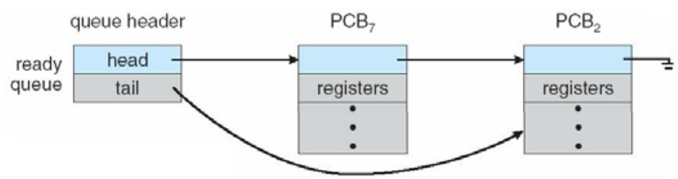
- A process can create several other process a.k.a child or sub processes
  - Exploit the ability of the system to concurrently execute
  - E.g. ***gcc*** program invokes different processes for the compiler, assembler, and linker
- Each process could get its resources directly from OS
  - Can restrict resources to subset of parent's process
    - Prevent overloading of system with too many children
- The new process gets its own space in memory
  - Parent and child processes address space are still different
- Because parent and child are isolated, they can communicate only via system calls

Process Management

- OS maintains a data structure for each process called *Process Control Block* (PCB)
- Information associated with each PCB:
  - Process state: e.g. ready, or waiting etc.
  - Program counter: address of next instruction
  - CPU registers: PC, working registers, stack pointer, condition code
  - CPU scheduling information**:** scheduling order and priority
  - Memory-management information: page table/segment table pointers
  - Accounting information: book keeping info e.g. amt CPU used
  - I/O status information: list of I/O devices allocated to this process

Ready Queue

- Processes resident in main memory and that in *ready* state are kept in a *ready queue*
  - Process waits in the ready queue until selected
- Unless a process terminates, it will eventually be put back into a ready queue



Similarly OS keeps device queues for processes waiting for I/O

Process Scheduling

- Idea of multi-tasking or multiprogramming
- Realize that process has cpu-burst and I/O burst cycle
  - When I/O burst, CPU idle
  - Exploit the idleness to better achieve parallel tasking
    - On Uniprocessor system switch between processes so fast to give an illusion of parallelism
- Determine which process should be next in line for the CPU
  - Selects from among the processes that are *ready* to execute (more on scheduling algorithms later)

Non-Preemptive vs. Preemptive

- A process can give up CPU in two ways
- Non-preemptive: A process voluntarily gives up CPU
  - I/O request
    - Process is blocked, then when request ready it is put back into ready queue
  - A process creates a new child/sub process (more later)
  - Finished Instructions to execute (Process termination)
    - PCB and resources assigned are de-allocated
- Preemptive: A process is forced to give up the CPU
  - Interrupted due to higher priority process
  - Each process has fixed time-slice to use CPU

Multithreading

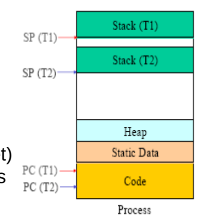
- Multithreading a program *appears* to do more than one thing at a time
- The idea of Multithreading is same as Multiprogramming i.e. multitasking but within a single process
  - Multiprogramming is multitasking across different process
- E.g. A word processing program has separate threads:
  - One for displaying graphics
  - Other for reading in keystrokes from the user
  - Another to perform spelling and grammar checking in the background

Why do Multithreading ?

- A process includes many things:
  - An address space (defining all the code and data pages)
  - OS descriptors of resources allocated (e.g., open files)
  - Execution state (PC, SP, regs, etc).
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
- Communicating between processes is costly because most communication goes through the OS
  - Inter-Process Communication (IPC)
  - Overhead of system calls and copying data

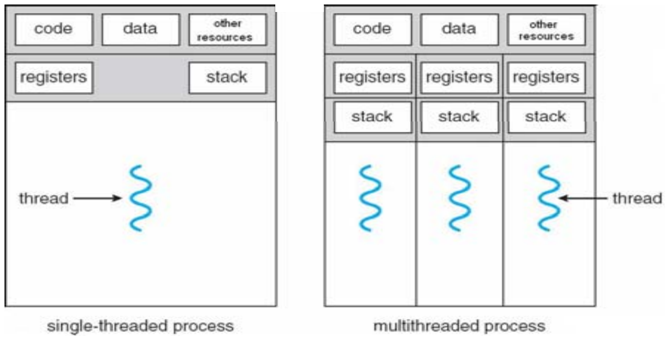
Multithreading (contd..)

- Allow process to be subdivided into different threads of control
- A *thread* is the smallest schedulable unit in multithreading
- A thread in execution works with
  - thread ID
  - Registers (program counter and working register set)
  - Stack (for procedure call parameters, local variables etc.)
- A thread *shares* with other threads a process's (to which it belongs to)
  - Code section
  - Data section (static + heap)
  - Permissions
  - Other resources (e.g. files)



Process with 2 threads

Difference between Single vs. Multithread Process



- A process by itself can be viewed a single thread and is traditionally known as a heavy weight process

Advantages of Multithreading

- Increase responsiveness to the user
  - Allows a program to continue running even if parts of it is "waiting"
- Resource Sharing
  - Threads share memory and resources of the process to which they belong
  - All threads run within same address space
- Economical
  - They can communicate through shared data and thereby eliminate the overhead of system calls
- Multiprocessor system
  - They allow you to get parallel performance on a shared-memory multiprocessor

Threads

- Like process states, threads also have states:
  - New, Ready, Running, Waiting and Terminated
- Like processes, the OS will switch between threads (even if though they belong to a single process) for CPU usage
- Like process creation, thread creation is supported by APIs
- Java Threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- In C, threads are created using functions in <pthread.h> library (p stand for posix)

Sharing Address Space

- Sharing address space requires only one copy of code or data in main memory
  - E.g.1: 2 processes share the same library routine (code)
  - E.g.2: A print program produces characters and that is consumed by printer driver (two processes sharing data)
  - E.g.3: Threads within a process share (global) data section
- As long as shared data is not being modified there is no problem
- But concurrent access to shared data that modify the value of the data can lead to *data inconsistency*
  - E.g. Printer driver consumed data before print program produced it

Threading Example

```
class Counter {
  private int c = 0;
  public void increment() {
   c++;
  }
  public void decrement() {
       c--;
   }
  public int value() {
       return c;
   }
 }
```

- If a Counter object is referenced from multiple threads
- There will be interference between threads when 2 operations (increment and decrement), running in different threads, but acting on the same data (i.e. c)
- This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Threading Example (contd..)

- Remember that single expression "c++" can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

- The same applies for c--

Threading Example (contd..)

- Suppose Thread 1 invokes increment at about the same time Thread 2 invokes decrement
  - In reality OS is going to switch between Thread 1 and 2
- If the initial value of c is 0, their interleaved actions might follow this sequence:

1. Thread 1: Retrieve c
2. Thread 2: Retrieve c
3. Thread 1: Increment retrieved value; result is 1
4. Thread 2: Decrement retrieved value; result is -1
5. Thread 1: Store result in c; c is now 1
6. Thread 2: Store result in c; c is now -1

Race Condition

- In previous example, Thread 1's result is lost, overwritten by Thread 2
  - Many different interleaving can result in different value of "c"
- Race Condition
  - Several process/threads access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place

Synchronization

- To avoid race conditions, need to guarantee "atomic" execution of sequence of instructions
  - Execute without an interruptions
  - Mutual exclusion for shared data
- Big Picture
  - Request entry to critical section (regions of code that may change shared data)
  - Access (shared) data in critical section
  - Exit from critical section

Implementing Synchronization

- OS
  - Done via system call
  - Block (wait) until you have exclusive access
  - Interrupts are temporarily disabled to carryout atomic execution
- Low-level support
  - E.g. Test-and-Set (TS) instruction provided in IBM/370 ISA
  - Lock/Unlock mechanism
    - Lock state is implemented by a memory location
    - Location contains value 0 if the lock is unlocked and 1 if the lock is locked
    - If value is 0, then lock is closed and critical section is executed. After finishing the critical section, the lock is opened
  - This support is usually for shared memory multiprocessor system
    - CPU executing such a instruction locks the memory bus to prohibit other CPUs

Synchronization Mechanics for Programmer

- High-Level Language constructs which inherently translates to OS system calls
  - E.g. In **Java** you can synchronize methods using ***synchronized*** keyword
  - Guarantees mutual exclusion i.e. acquires the intrinsic lock for that method's object and releases it when method returns
  - Guarantees that changes to the state of the object are visible to all threads

```
public class SynchronizedCounter {
 private int c = 0;
public synchronized void increment() {
       c++;
}
public synchronized void decrement() {
       c--;
}
public synchronized int value() {
       return c;
 }
```