

# System Software – Part I

CIT 595  
Spring 2007

## What is System Software?

- **System Software** is a generic term referring to any computer software which manages and controls the hardware so that application software can perform a task
- It is an essential part of the computer system
  - Allows *users* to interact with the system to enhance productivity
    - E.g. Word Editor, Spreadsheet programs etc.
  - Allows *programmers* to develop their own applications without knowing too much detail of the underlying hardware
    - E.g. High-level programming languages and tools

CIT 595

13 - 2

## System Software

System Software includes:

- Operating System
  - Help various applications interact with computer hardware
- Programming Tools
  - Help programmers in application development with aid of system utilities such as assemblers, compilers, linkers etc.
- Middleware (not our focus)
  - Above the OS but below the application program layer
  - Used to support distributed applications
  - E.g. Allow programs written for access to a particular database to access other databases

CIT 595

13 - 3

## What is an Operating System?

- An Operating System is a program that manages the computer hardware
- Goal is to provide:
  - *Efficiency* in terms of resource usage (CPU, Memory, I/O)
  - *Convenience* in terms of user usability

CIT 595

13 - 4

## Operating System Evolution

- Earlier OS known as *resident monitors* allowed Monoprogramming environment
  - Loaded a program
  - Gave control of the CPU to the program
  - Regained control once program was done executing
- OS evolved into allowing Multiprogramming and Time-sharing environment
  - CPU usually idle when waiting for disk, tape, or network access
  - Utilize idle time for other programs
  - Utilize idle time of one user to service another user request in system shared multiple users

CIT 595

13 - 5

## Multiprogramming and Time-Sharing

- *Multiprogramming* environment allows several executing programs to be in memory concurrently
  - This is possible due to Virtual Memory
  - Cycle through programs (processes) such time each one gets to use CPU for a specific time slice
  - Achieves maximum processor utilization
- *Time-sharing* allows multiple users to share a system
  - E.g. Airline Reservation System
  - Possible as interactive system use often spend much of their time idly waiting for user input/retrieving data from a device
  - Cycle through users giving each user a small slice of the processor time
  - Achieves minimum response time for the user

CIT 595

13 - 6

## Services of Operating System

- Interface manager
  - Human interaction made easy for average user
  - Abstraction, control and sharing for programmers
- Resource manager
  - Efficient use of resources (CPU, Memory, and I/O)
- Process Management, Scheduling and Synchronization
- Enhances hardware features
  - Virtual Machines
- Data security and protection
  - Manages execution of user programs to prevent errors and improper use of the computer

CIT 595

13 - 7

## Aside: Where is the O.S when Processor starts ?

- Before the O.S can perform any of its services it itself has to be loaded into main memory
  - Remember that main-memory is volatile
  - So OS software has to reside somewhere on the disk
- CPU needs to know where OS program is located so that the OS can start performing its services
  - The BIOS (Basic Input Output System) software loads the operating system at boot (starting) time
  - BIOS software is stored on non-volatile memory Read Only Memory (ROM)
  - Basically CPU executes code from ROM to start the OS

CIT 595

13 - 8

## Processor-OS Interaction in Mutiprogramming

- OS must have control of the processor as one of it many task is to schedule programs (processes) that use the CPU
- It gives up the CPU to various processes during the course of their execution
- OS regains the control of CPU whenever a process wishes another resources
  - E.g. I/O device such as disk
  - Hence all I/O operation is setup through OS system call
  - Not only allows protection but a way to regain control of the processor

CIT 595

13 - 9

## OS Service: Human Interface

- Provides layer of abstraction between the user and the hardware of the machine
- Provides to kinds of Interfaces:
  - *Command Line*: provided with command prompt and commands that will allow the user to interact with the system
    - Disadvantage: know the syntax of the actual system
  - *Graphical User*: graphical representation of the command line interface
    - E.g. Drag drop files, Directory Structure viewable as folders

CIT 595

13 - 10

## OS Service: Resource Management

- Multiple processes share *CPU*
  - Access to CPU is managed by *OS scheduler*
  - Works in tandem with Process Management
- Multiple processes share the same *physical/main memory*
  - OS manages transparent movement of data between physical memory and permanent storage devices – illusion of *Virtual Memory*
  - Before VM, programmer was responsible for memory management (overlay technique)
- Multiple process share same *I/O resources*
  - Allow generic interface for I/O through various *system calls*
  - Also manages file system and their mapping on the secondary storage devices such as disks
  - More on this when we study I/O (chp 7)

CIT 595

13 - 11

## OS Service: Process Management and Scheduling

- Process: An instance of program in execution
- In multiprogramming environment, the OS allows several processes to be in memory concurrently and time share the CPU
- Goal of multiprocessing is to have a process running on the CPU at all times
- Need to *manage and schedule processes* such that we achieve maximum CPU utilization

CIT 595

13 - 12

## Managing a process

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

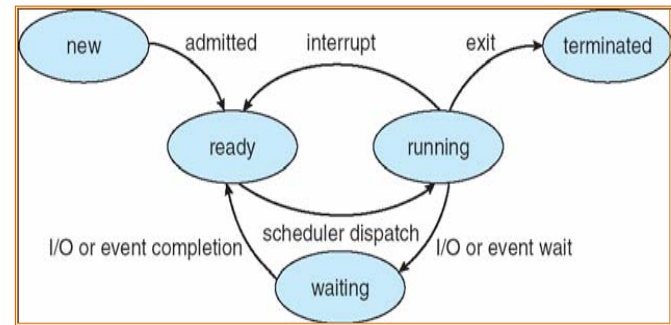
State/Activity	Description
new	The process is being created
running	Instructions are being executed on the processor
waiting	The process is waiting for some event to occur
ready	The process is waiting to be assigned to processor
terminated	The process has finished execution

Note: state names vary across different OS

CIT 595

13 - 13

## State Diagram of a Process Activity/State



Note: That only one process can be in running state while many processes can be in ready and waiting states

CIT 595

13 - 14

## How does OS track each process?

- 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
  - Accounting information: book keeping info e.g. amt CPU used
  - I/O status information: list of I/O devices allocated to this process and files (open)

CIT 595

13 - 15

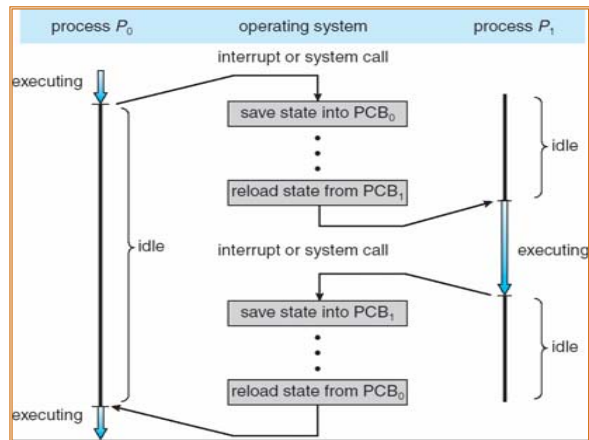
## Context Switch

- Switching between processes is known as Context Switch
- During context switch all *pertinent information* about the current executing process must be saved
- So that when the process gets the CPU again, it can start from where it left off
- Hence on every context switch the PCB of the process giving up the CPU is updated

CIT 595

13 - 16

## CPU Switch From Process to Process



CIT 595

13 - 17

## Process Scheduling

- The idea is to switch between processes so frequently such that users feel they are interacting with multiple programs concurrently
- OS has to schedule processes appropriately such that no process:
  - Keeps the CPU for itself for too long
  - Be starved or deprived from other system resources such as memory, and I/O

CIT 595

13 - 18

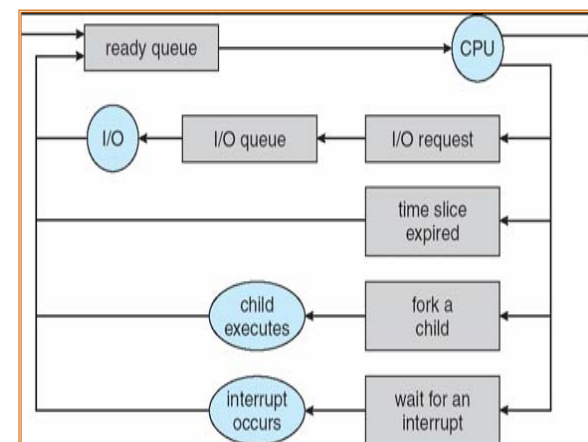
## Scheduling Queue

- Processes resident in main memory and that in *ready* state are kept in a *ready queue*
- Process waits in the ready queue until selected
- Once the process is assigned a CPU it can continue until:
  - It issues I/O request
  - Forcibly removed either due to an interrupt or its time slice is up
  - Create a new sub process (known as child process) – more on this later
- Unless a process terminates, it will eventually be put back into a ready queue
  - When the process terminates it is removed from queue and its PCB and resources assigned to are de-allocated

CIT 595

13 - 19

## Ready Queue Representation

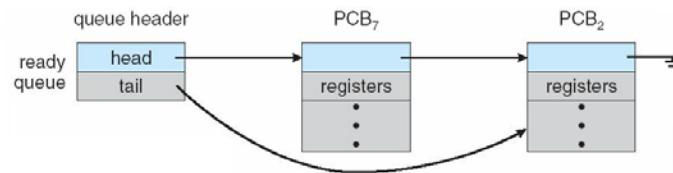


CIT 595

13 - 20

## Ready Queue Implementation

- A ready queue is linked list data structure
  - Ordering depends on the scheduling algorithm
- The records in the queue are PCB's
  - The head and tail of the linked list point to first and final PCB in the list
  - Each PCB includes information of the next ready process in the ready queue



CIT 595

13 - 21

## OS Scheduler

### Short Term Scheduler

- Selects from among the processes that are *ready* to execute and allocates one of them to CPU
- Invoked over a short duration of time e.g. every 100 milliseconds
- Because of the short durations between invocations, the scheduler must be fast
  - E.g. If it takes 10 ms to decide access the situation and decide the order of execution of the processes then  $10/(100 + 10) = 9\%$  of the CPU is being used simply for scheduling the work

CIT 595

13 - 22

## Non-Preemptive vs. Preemptive Scheduling

- A process can be scheduled to give up CPU in two ways
- Non-preemptive: A process voluntarily gives up CPU
  - I/O request
  - Finished Instructions to execute (Process termination)
- Preemptive: A process is forced to give up the CPU
  - Each process has fixed time-slice to use CPU
  - Interrupted due to higher priority process

CIT 595

13 - 23

## Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that completed per time unit
- Turnaround time – amount of time to execute a particular process
  - Waiting in Ready Queue + Executing on CPU + doing I/O
- Waiting time – amount of time a process has been waiting in the ready queue (Most algorithms decide to lower Waiting Time)
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not the time it takes to output the response** (for time-sharing environment)

CIT 595

13 - 24

## Scheduling Scheme: FCFS

First Come First Served (FCFS)

- The process that requests the CPU first is allocated the CPU
- Implemented using a FIFO queue
  - When the process enters the read queue, its PCB is linked to the tail
  - The process at the head of the queue is given to the CPU
- FCFS is non-preemptive and hence easy to implement
- Not ideal for reducing Waiting Time
  - Wait Time vary substantially if the processes that come first are CPU intensive

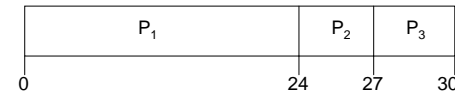
CIT 595

13 - 25

## FCFS Example (Execution Time)

Process	Execution Time
$P_1$	24
$P_2$	3
$P_3$	3

Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt/Time Chart for the schedule is:



Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$   
Average waiting time:  $(0 + 24 + 27)/3 = 17$

CIT 595

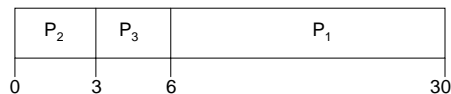
13 - 26

## FCFS Example (contd..)

Suppose that the processes arrive in the order

$P_2, P_3, P_1$

The Gantt/Time chart for the schedule is:



Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$   
Average waiting time:  $(6 + 0 + 3)/3 = 3$

Much better than previous case

Convoy effect: all processes wait for the one big process to get off the CPU

CIT 595

13 - 27

## Scheduling Algorithm: Shortest Job First (SJF)

- The process with the shortest execution time takes priority over others
- Associate with each process the length of its next CPU execution time
  - Achieved by **guessing** the run time of process for its next execution
- Use these lengths to schedule the process with the shortest time
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU execution time is complete. Also if they are two short jobs with same execution time then use their time of arrival to break the tie
  - **Preemptive** – if a new process arrives with CPU execution time less than remaining time of current executing process, preempt

CIT 595

13 - 28

### SJF with Preemption Example

Process	Arrival Time	Next CPU Ex Time
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

SJF (preemptive)



$$\text{Average waiting time} = (9 + 1 + 0 + 2)/4 = 3$$

Also known as Shortest Remaining Time First

CIT 595

13 - 29

### SJF Advantages and Disadvantages

Advantage

- SJF is optimal – gives minimum average waiting time for a given set of processes

Disadvantage

- Need to have a good heuristic to guess the next CPU execution time

CIT 595

13 - 30

### Estimating the Next CPU Execution Time

- Use previous value of the execution time
- Predict new time using Exponential Averaging
- Method:
  1.  $t_n$  = actual length of  $n^{\text{th}}$  CPU Execution Time
  2.  $\tau_{n+1}$  = predicted value for the next CPU Execution Time
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Formula:  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

CIT 595

13 - 31

### Scheduling Scheme: Priority

- A priority number (integer) is associated with each process
- CPU is allocated to the process with the highest priority
  - If processes have same priority then schedule according to FCFS
- SJF is an example of a priority scheduling where priority is the (predicted) is the shortest next CPU execution time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses increase the priority of the process

CIT 595

13 - 32

## Scheduling Scheme: Round Robin (RR)

- Each process gets a small unit of CPU time called *time quantum or slice*
  - After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once
  - No process waits more than  $(n-1)q$  time units
- Performance
  - If  $q$  is large then RR becomes like FCFS
  - $q$  should not be so small such that it requires too many context switches
    - Slows down the execution of the process

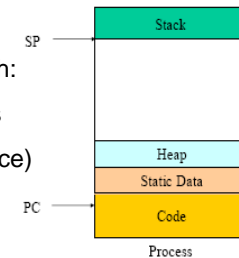
CIT 595

13 - 33

## Recap: 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)



CIT 595

13 - 34

## 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
  - Create a subtask to do something else
  - E.g. **gcc** program invokes different processes for the compiler, the assembler, and the linker
- Each process could get its resources directly from OS
  - OS can restrict resources to subset of original/parent process to prevent overloading of system with too many child processes
- The parent may also pass along some data to child process upon creation
  - E.g. In gcc, compiler produces assembly code that is passed onto the assembler
  - However communication between processes is restricted only via system calls, that is OS will intervene to pass the information

CIT 595

13 - 35

## Process Creation

- The new process gets its own space in memory
  - Hence parent and child processes address space are still different – for process isolation
- The memory of the new process is either loaded with:
  - Copy of the program the parent process is running
    - E.g. If the parent process was Spreadsheet Program
    - Then child is another instance of the spread sheet program but with different data e.g. different file name
  - Another program that parent wishes the sub process to execute
    - The parent has to tell OS which program that might be (again this done via system call)

CIT 595

13 - 36

## Process Creation Ability

- Since processes in the system can execute concurrently
- OS must *provide a mechanism* for process creation as well as process termination at the user level
  - Usually done through an API or system call
- Example: In Unix OS
  - Process creation is done using *fork()* system call
    - *execlp()* system call always a different program to be loaded into memory space of the child process
  - The process can wait for child to complete using the *wait()* system call (but sometimes it may continue concurrently)
  - There more functions as well under <unistd.h>

CIT 595

13 - 37

## Example of Process Creation using C

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.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 */
        //Overlay address space with UNIX command "ls"
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL); //NULL = 0 (MACRO)
        printf("#####\n");
        printf ("Child Complete\n");
        exit(0);
    }
}
```

CIT 595

Note: Run the code on eniac-s

13 - 38

## Process Termination

- A process terminates when it finishes executing its final statement and asks the OS to delete it using the *exit()* system call
- If a child process terminates then it may return data to its parent process via the *wait ()* system call
- When a process terminates all the resources such as memory and I/O are de-allocated by OS

CIT 595

13 - 39

## Multithreading

- In Multithreading a program *appears* to do more than one thing at a time
- The idea of Multithreading is same as Multiprocessing i.e. multitasking but within a single process
  - Multiprocessing is multi-tasking 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

CIT 595

13 - 40

## Multithreading (contd..)

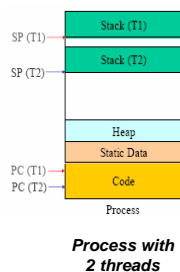
- Allow process to be subdivided into different threads of control

- 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)

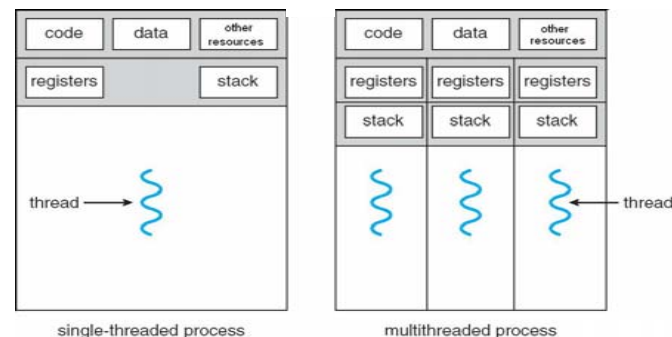
- code section
- data section (static + heap)



CIT 595

13 - 41

## 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

CIT 595

13 - 42

## Advantages of Multithreading

- Responsiveness
  - Allows a program to continue running even if part of it is "waiting"
  - Increase responsiveness to the user
- Resource Sharing
  - Threads share memory and resources of the process to which they belong
  - All threads run within same address space
- Economical
  - Threads are more economical to create due to sharing of memory and resources
  - Also the context switch time overhead is less in thread creation

CIT 595

13 - 43

## Advantages of Multithreading (contd..)

- Utilization of Multiprocessor Architectures
  - So far we worked with unprocessed architecture i.e. one CPU
  - CPU generally moves between each thread so quickly such that it creates an illusion of parallelism
  - In multiprocessor architecture i.e. more than one CPU, each thread can may be running in parallel on different processor

CIT 595

13 - 44

## Thread Creation

- Like process creation, thread creation is supported by APS
- E.g. Java Threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

CIT 595

13 - 45

## Threads and Processes

- Like process states, threads also have states:
  - New, Ready, Running, Waiting and Terminated
- In uniprocessor system, like processes, the OS will switch between threads (even if though they belong to a single process) for CPU usage

CIT 595

13 - 46

## Sharing Address Space

- Sharing address space requires only one copy of code or data in main memory
  - E.g.1: Two 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
  - More detail example with Threading on next slide

CIT 595

13 - 47

## 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
- E.g. Interference happens when two 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.

CIT 595

13 - 48

### 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--

CIT 595

13 - 49

### 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

CIT 595

13 - 50

### Race Condition

- In previous example, Thread 1's result is lost, overwritten by Thread 2
  - Many different interleavings can result in different value of "c"
- A situation where 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 is known as *Race Condition*

CIT 595

13 - 51

### Synchronization

- Hence to avoid race conditions, no thread/process should be guaranteed exclusive access to the shared data at any point in time
- Safely allowing *mutually exclusive* access to shared data requires *synchronization*

CIT 595

13 - 52

## How to Synchronize Access to shared data?

- Regions of code that may change shared data is called *critical-region*
  - Increment() and Decrement() in Counter Example
- Need to ensure that only 1 process or thread accesses shared variable until the required sequence of operations in critical region has been completed
- Synchronization protocol for each of n processes/threads using the shared section:
  - request entry to critical section
  - access (shared) data in critical section
  - request exit from critical section

CIT 595

13 - 53

## Synchronization Protocol

- The protocol described (previous slide) can be conceptualized as *locking* mechanism
  - When one thread/process is in critical-section, the region is locked for the other threads/processes
  - Only can enter critical section if lock is open
- The lock state is implemented by a memory location
  - The location contains value 0 if the lock is unlocked and 1 if the lock is locked
  - This is e.g. with two threads/processes can be further expanded to multiple threads/processes
  - The update to this memory location is atomic (no interruptions) and is guaranteed by the OS

CIT 595

13 - 54

## Implementation of Synchronization

- Synchronization feature is also provided by OS through System Call
  - wait()
    - Check if critical-section is accessible?
    - If not accessible (i.e. lock = 1), then process/thread is added to wait queue (blocked)
    - else allowed it to enter critical section
  - signal()
    - Once process is done with critical-section
    - Gives up access (or lock), then process that is waiting in queue is informed

CIT 595

13 - 55

## Synchronization Mechanics for Programmer

- Through 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;
    }
}
```

CIT 595

13 - 56

## Deadlock in Resource Sharing Environment

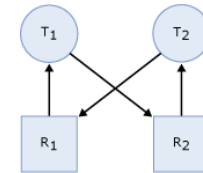
- A *deadlock* occurs when 2 or more processes/threads permanently block each other by each having a lock on a resource which the other tasks are trying to lock
- Deadlock can occur due to
  - Locks: Waiting to acquire locks on resources, such as objects, pages etc.
  - Sharing resources such as I/O devices printer, disks etc.

CIT 595

13 - 57

## Deadlock High-Level View

- Task (thread/process) T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).



- Task T2 has a lock on resource R2 and has requested a lock on resource R1
- Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists

CIT 595

13 - 58

## Deadlock Detection

- OS must also monitor or avoid deadlock
- Monitor (more practical approach):
  - Another service routine that periodically goes and checks on waiting tasks
  - If deadlock is determined, then
    - Abort all processes/threads within that are deadlock – expensive
    - Abort one process at a time till deadlock is cycle is eliminated
- Avoidance:
  - By not explicitly over allocating resources
  - However this requires prior knowledge of total resources needed by each task – not practical
  - Difficult but many algorithms are proposed e.g. Banker's Algorithm

CIT 595

13 - 59

## Operating System Service: Security and Protection

- Concurrent processes are protected from each other (may it user processes or OS processes) by Virtual Memory
- Two modes of operation: user and supervisor, to prevent unauthorized use
- Resource protector: Request for resources by a processes goes through OS
  - E.g. I/O operation is done via system call, hence OS intervenes

CIT 595

13 - 60

## Operating System Design

- OS are fundamentally *event-driven* systems – i.e. wait for an event to happen, respond appropriately to the event, then wait for the next event. Example:

- A user program issues a system call to read a file
- The OS figures out which disk blocks to bring in, and generates a request to the disk controller to read the disk blocks into memory
- The disk controller finishes reading in the disk block and generates an interrupt. The OS moves the read data into the user program and restarts the user program.

- O.S is also called the kernel. There are two design philosophies with kernels:

- Microkernel
- Monolithic Kernels

CIT 595

13 - 61

## Microkernel Design

- Provide basic OS functionality
  - Minimum process and memory management
- Rely on other modules to perform specific tasks
- Moves many OS services into user space
  
- Main task is to provide communication between program and the various services (also in user space)
- Advantages
  - If new services are added, microkernel does not need to be changed
  - Kernel is small and portable from hardware to another – very limited change to kernel program
  - Provides security as services are running at user level access
- Disadvantage
  - Communication between kernel and other modules is necessary, which makes the system slower and less efficient

CIT 595

13 - 62

## Monolithic Kernel

- Provide all the functionality through one single process
- Interacts directly with the hardware
- No communication required
- Size of the Kernel program is very large compared to microkernel approach
- Not easily portable
- E.g. Linux, MacOS

CIT 595

13 - 63