CPE 545

Protocol Software

Protocol Software

- Specification involves:
  - Architectural relationship of communicating entities
    - Client/server, publisher/subscriber, node-to-node
  - Valid execution states for each entity
    - Initialized, connected, waiting for event, Listening , Message Sent, Message Received, disconnected, …
  - Actions to be taken when events (messages) are triggered (received)
  - Protocol Data Units for communication between entities
    - Message format
  - Timers used by the entities

Protocol Software

- Defines the language used for communicating systems and can be defined at each layer of OSI
- Protocol Implementation
  - Protocols can be standards based (IEEE, ANSI) or custom and proprietary
  - Two important reasons for proprietary protocols
    - Standards based protocol insufficient for application
    - Intellectual property protection for competitive advantage
  - Specification can be defined using protocol specification languages

Protocol Implementation: State Machines

- Stateless protocol
  - Current state of protocol is independent of the actions in the previous state (IP Forwarding)
- Stateful protocol
  - Current state of protocol depends on the previous state and the sequence of actions in that state (TCP)
  - Use state machines to specify various states, the transition events and actions to perform
  - Valid events can be protocol messages or timer events

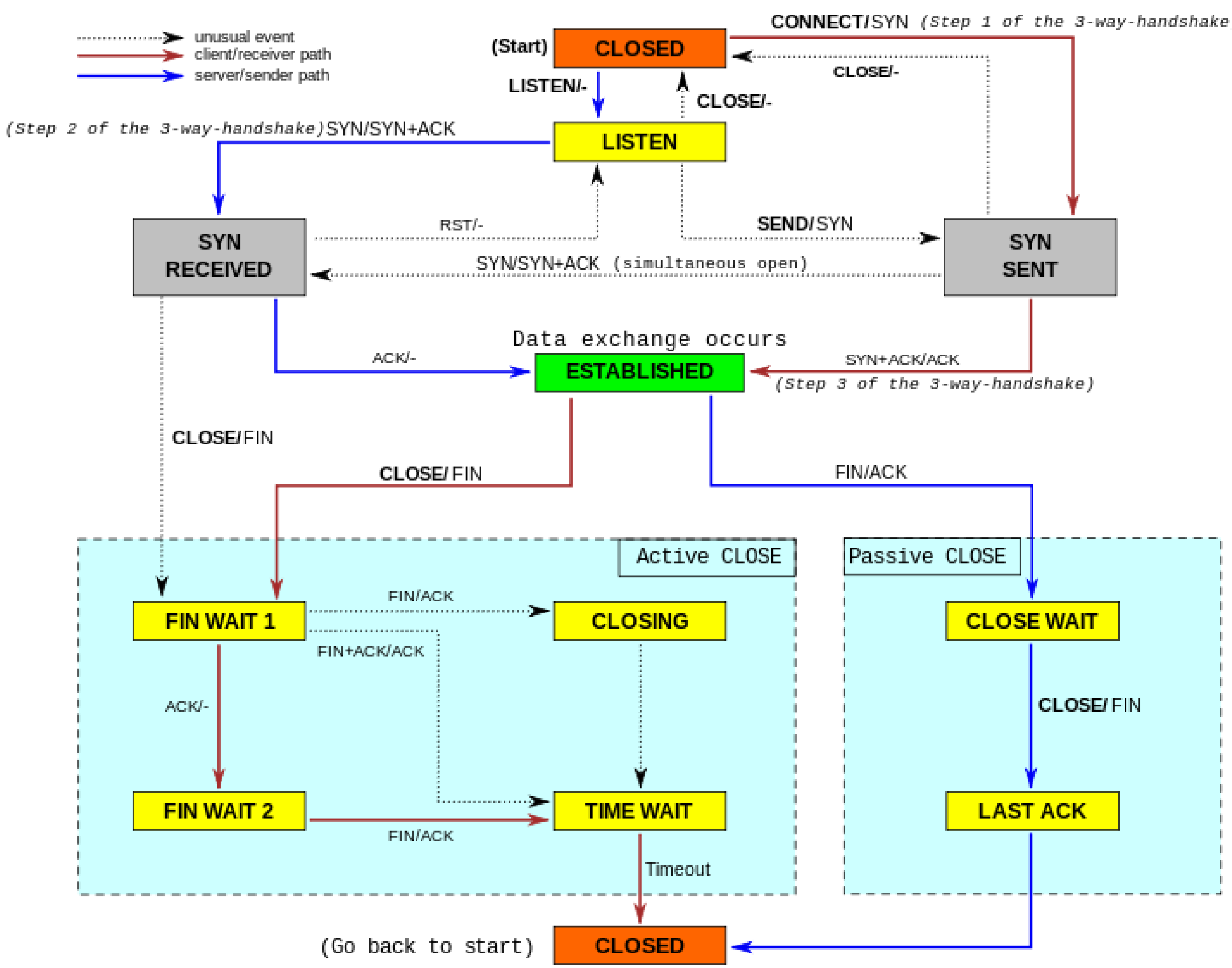
Protocol Implementation: State Machines (TCP)

- LISTEN
  - (server) represents waiting for a connection request from any remote TCP and port.
- SYN-SENT
  - (client) represents waiting for a matching connection request after having sent a connection request.
- SYN-RECEIVED
  - (server) represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
- ESTABLISHED
  - (both server and client) represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.
- FIN-WAIT-1
  - (both server and client) represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
- FIN-WAIT-2
  - (both server and client) represents waiting for a connection termination request from the remote TCP.

Protocol Implementation: State Machines (TCP)

- CLOSE-WAIT
  - (both server and client) represents waiting for a connection termination request from the local user.
- CLOSING
  - (both server and client) represents waiting for a connection termination request acknowledgment from the remote TCP.
- LAST-ACK
  - (both server and client) represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).
- TIME-WAIT
  - (either server or client) represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request. [According to RFC 793 a connection can stay in TIME-WAIT for a maximum of four minutes known as a MSL (maximum segment lifetime).]
- CLOSED
  - (both server and client) represents no connection state at all.

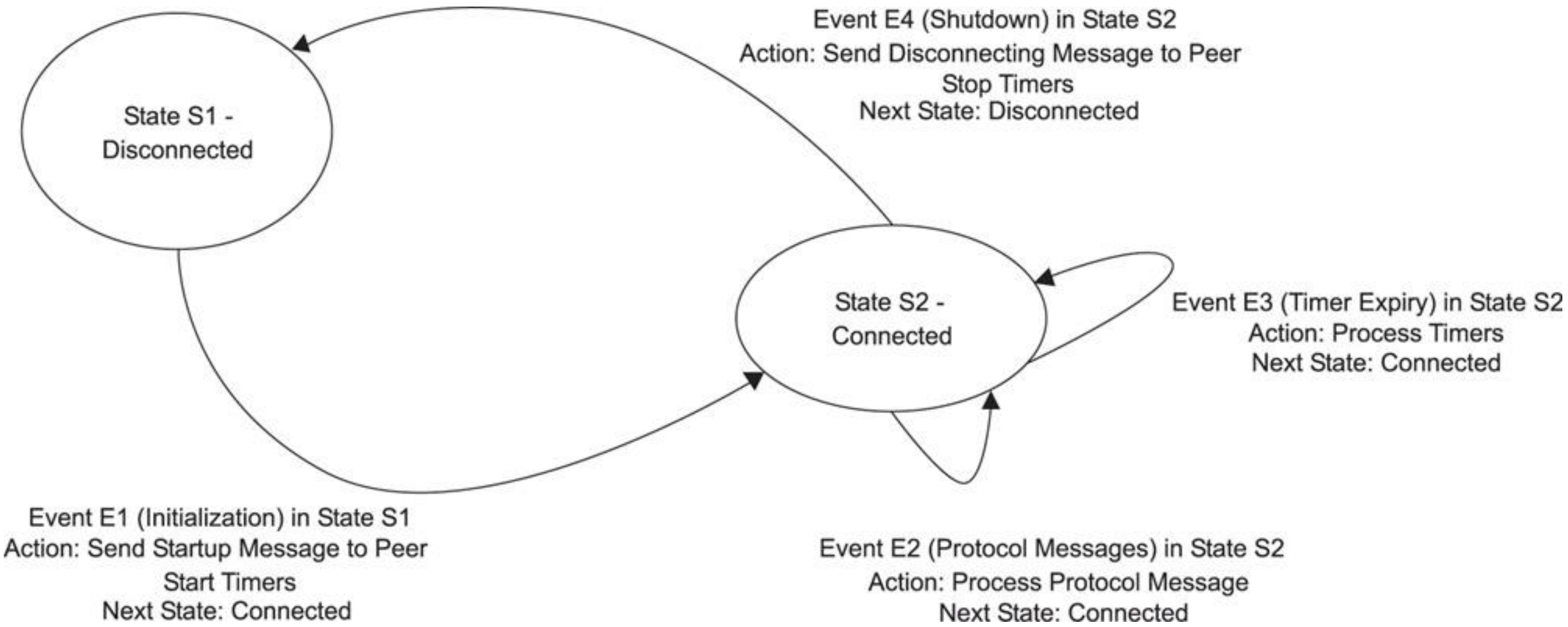
Protocol Implementation: State Machines (TCP)



Protocol Implementation: State Machines

- State machine implementation
  - Simplest way is to use a switch-case statement construct
    - Difficult to implement when state machine is complex
  - State Event Table (SET) is an alternate method
    - Is a matrix where columns represent states and rows represent events
    - An individual entry in the matrix is a tuple {Action, Next State} indicates what action to perform and what state to move
    - are usually implemented as a library of functions that can be invoked by specific layers

Protocol Implementation: State Machines



Protocol Implementation: State Machines – Switch statement

```
switch (event)
{
	case E1: /* Initialize */
					If (current_state == DISCONNECTED)
					{
						 InitializeProtocol ();
						current_state = CONNECTED;
					} break;
	case E2: /* Protocol Messages */
					If (current_state == CONNECTED)
					{

					}
					break;
	case E3: /* Timer Event(s) */
					If (current_state == CONNECTED)
					{
						ProcessTimers ();
					}
					break;
	case E4: /* Disconnect Event */
					If (current_state == CONNECTED)
					{
						ShutdownProtocol ();
						current_state = DISCONNECTED;
					}
					break;
			default:
					logError ("Invalid Event, current_state, event);
					break;
}
```

Protocol Implementation:
State Machines – State Event Table (SET)

	S1	S2	S3
E1	{Action, Next State}	{Action, Next State}	{Action, Next State}
E2	{Action, Next State}	{Action, Next State}	{Action, Next State}
E3	{Action, Next State}	{Action, Next State}	{Action, Next State}
E4	{Action, Next State}	{Action, Next State}	{Action, Next State}
E5	{Action, Next State}	{Action, Next State}	{Action, Next State}

Protocol Implementation: State Machines – Set Tables

	Disconnected	Connected
E1 (Initialize)	InitializeProtocol() CONNECTED	Invalid Event
E2 (Protocol Messages)	Invalid Event	ProcessMessages (); CONNECTED
E3 (Timer Events)	Invalid Event	ProcessTimers (); CONNECTED
E4 (Disconnect)	Invalid Event	ShutdownProtocol() DISCONNECTED;

- protocol messages may be received from an external entity which may or may not be considered an error

State Machines – cont'd…

- Typical Events
  - messages
  - timer events
  - max retransmission attempts
  - error conditions
  - user intervention conditions

- Actions:
  - No-op or error routine action invoked when an event is invalid within the current state
  - All abnormal behavior of protocol should be identified and handled using actions upfront before system is deployed
  - Action routines can involve the construction of a new message for transmission (new event)

State Machines – cont'd…

- Predicates (Additional entry in SET)
  - The tuple will now be {Action, New State, *Predicate*}
  - Serves as an input to the action routine to help determine the course of execution within the routine
- Multiple State machines
  - Protocol can contain and often does contain multiple SET's
  - Needs to be a clear separation between the SET's

State Machines – cont'd…

```
While (1){
        Wait for any of the events;
         /* break out of the hard wait loop */
         if (MessageQueuing event)
              Process MessageQueue;
         If (timerEvent)
              Process TimerEvents
         Perform Housekeeping functions; /* e.g. release
                                    transmit buffers */
    }
```

State Machines – cont'd…

```
ProcessMessageQueue ()
        {
                    Determine type of message;   // What Message – PDU pre-processing
                    Classify the message and set the event variable;  // Map message to
Event
                    Pass event through the SET ; /*state machine access function*/
        }


        ProcessTimers ()
        {
                    Determine attributes of expired timers;  // What timer
                    Classify the timer type and set the event variable;  // Map timer to Event
                    Pass event through the SET; /*state machine access function*/
        }
```

Homework

- Describe what software components you would implement for an stateful protocol that employs multiple tasks in its design?

State Machines – contd…

- Entry for current state and event is:
  - SET [Event][CurrentState]

```
Ret = PerformAction (SET [Event][CurrentState]);
CurrentState = NextState (SET [Event][CurrentState]) ;

// Define a pointer to a function which is taking two  floats and returns an int

typedef int (*ActionFunction)(floats, floats);

typedef enum (STATE1, STATE2, ….} State;
typedef enum (EVENT1, EVENT2, ….} Event;


typedef struct
{
   ActionFunction  action;
   State           nextState;
} Tuple;
```

State Machines – contd…

```
Tuple setTable[EVENT_NUM][STATE_NUM] =
                {{foo1(float,float}, STATE3},
                  foo2(float,float}, STATE1},
                   ….
                  }

currentState = STATE1;

While (1)
{
      //Wait for message
            WaitForEvent(Msg * msg, ….)
            setTable[msg->eventType][currentState].action(f1,f2)
            currentState = setTable[msg->eventType][currentState].nextState;
}
```

Protocol Implementation

- PDU processing
- Memory management
- Buffer management
- Timer management
- Event Management
- IPC
- protocol and driver interface
- Protocol Management

Protocol Implementation: PDU Processing & Protocol Interfaces

- PDU’s are typically pre-processed before being passed to SET. Preprocessing involves:
  - Syntax verification of a packet
  - Checksum validation
- PDU’s are transmitted by the action routines of SET

Memory Management

- Required for allocating and releasing memory for applications in system heap

- Real time systems can have multiple memory partitions

  - Packet buffers can be maintained in DRAM partition while tables could be maintained in SRAM partition,

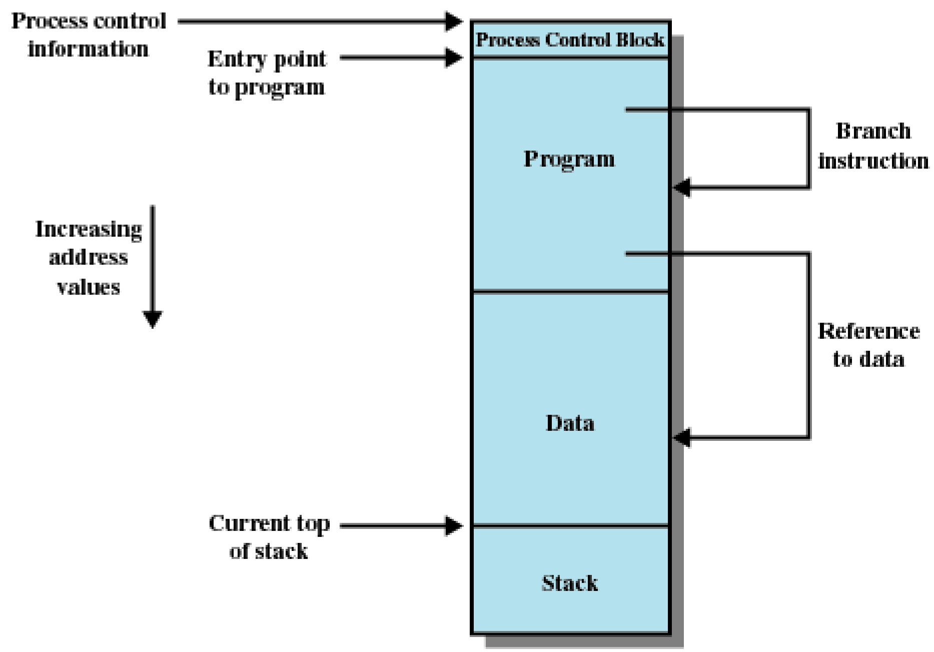
  - Each partition with their own memory management functions:

    - VxWorks™ RTOS, partitions can be created with the memPartCreate() call.

    - Individual blocks can be created out of these partitions with the routine memPartAlloc() and released with memPartFree().

Buffer Management

- Includes initialization, allocation
- Maintenance and release of buffers
- There can be multiple buffer pools, each consisting of buffers of a specific size.
- Memory management functions can be used for buffers
- Buffer management libraries are provided along with the RTOS—like the mbuf and zbuf libraries available in VxWorks.


Process control information
Entry point to program
Process Control Block
Program
Branch instruction
Increasing address values
Reference to data
Data
Current top of stack
Stack

Memory Management Requirements

- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Must be checked at run time
  - Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
- In embedded model, each process could be thought as the software on a single processor thus having multiple processors. Memory Protection is done by MPUs.

Memory Management Requirements

- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process access to the same copy of the program rather than have their own separate copy

Memory Management Requirements

- Logical Organization
  - Programs are written in modules
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules among processes

Fixed Partitioning

- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
  - If all partitions are full, the operating system can swap a process out of a partition
  - Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

Dynamic Partitioning Placement Algorithm

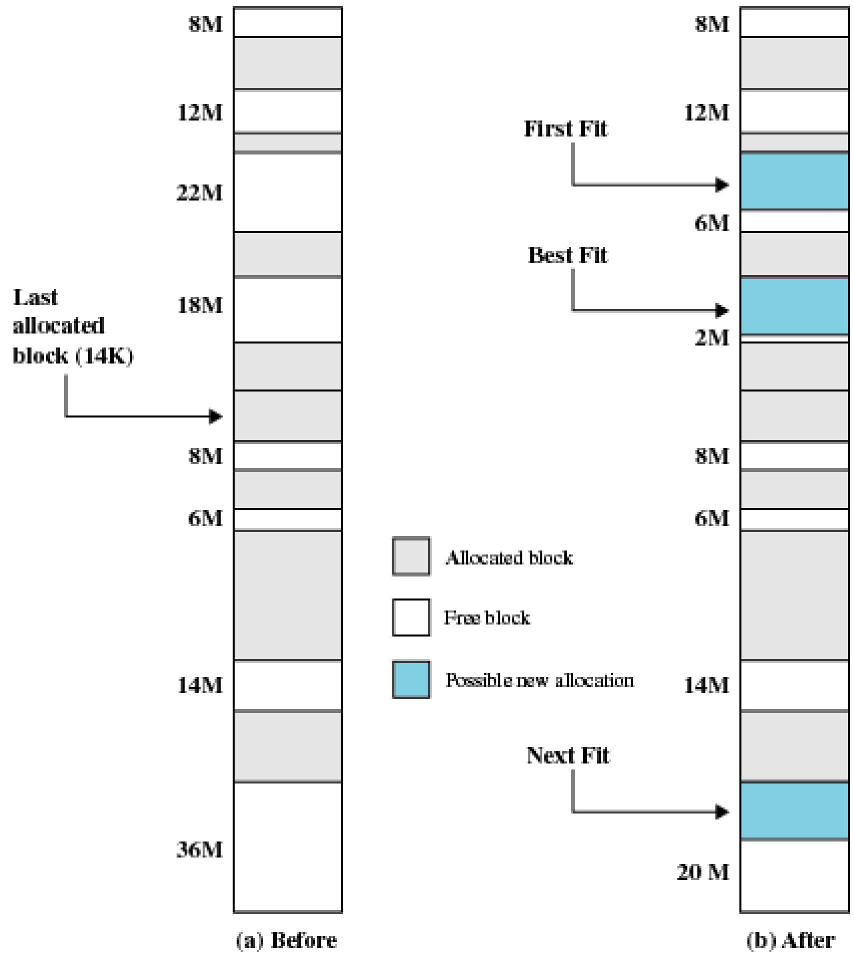
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left

Dynamic Partitioning Placement Algorithm

- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - More often allocate a block of memory at the front of memory

Dynamic Partitioning Placement Algorithm

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks



Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s <= 2^U$, entire block is allocated
  - Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to s is generated

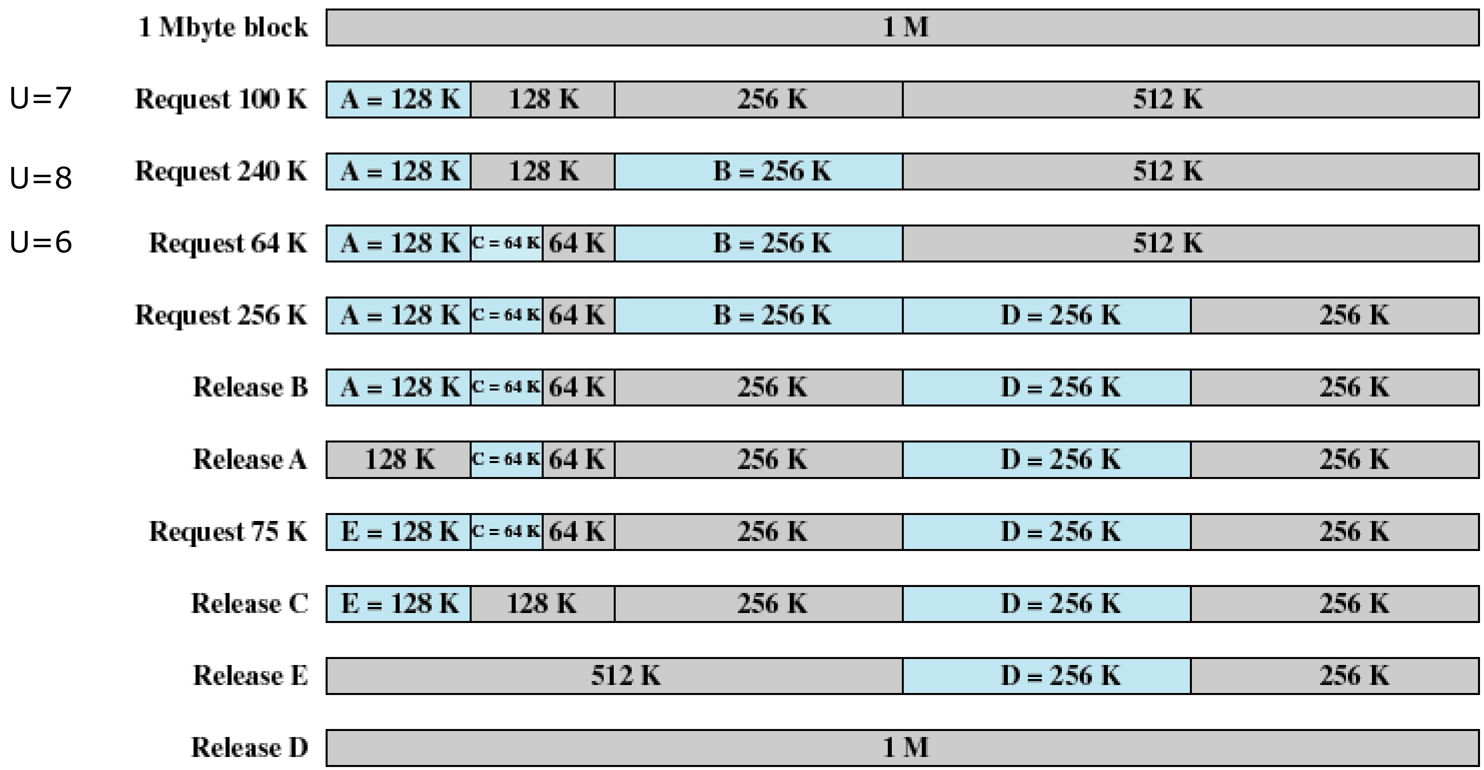
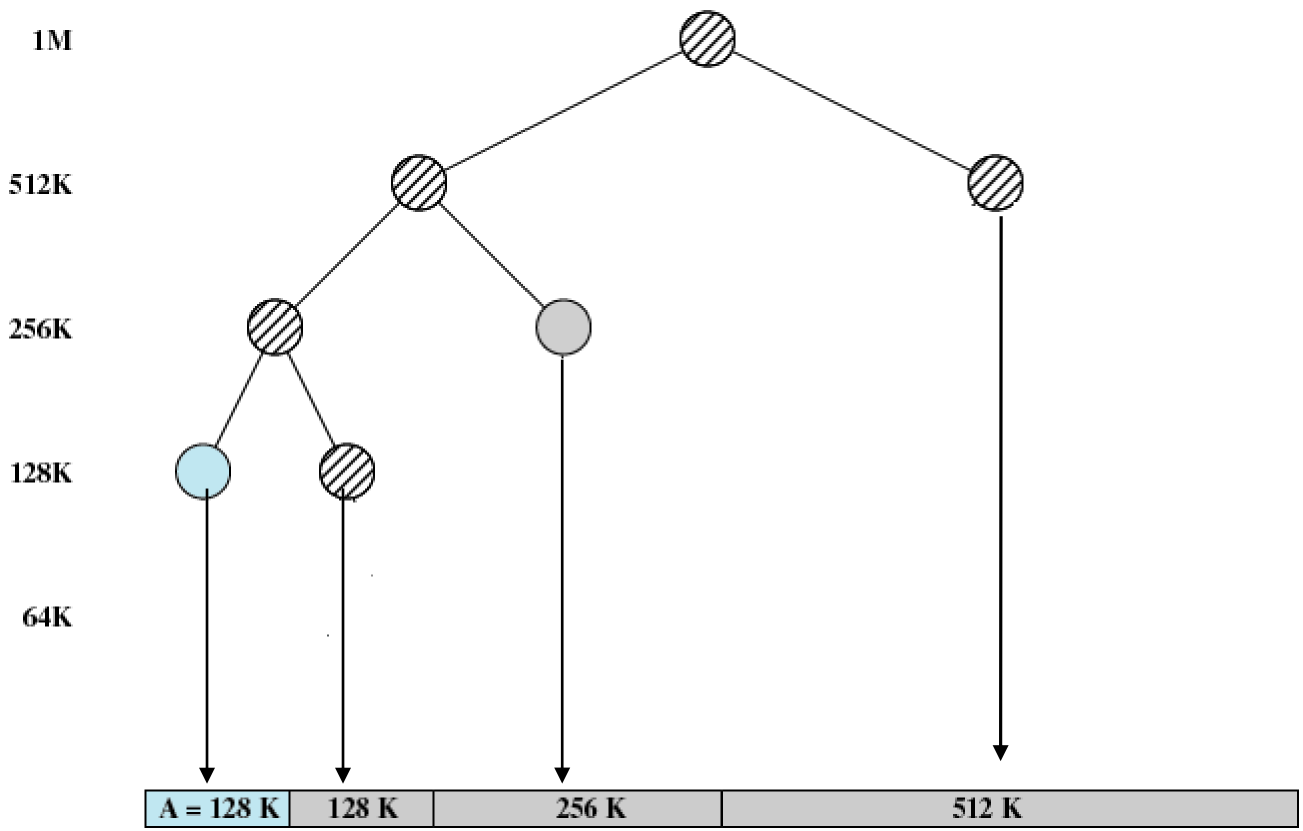
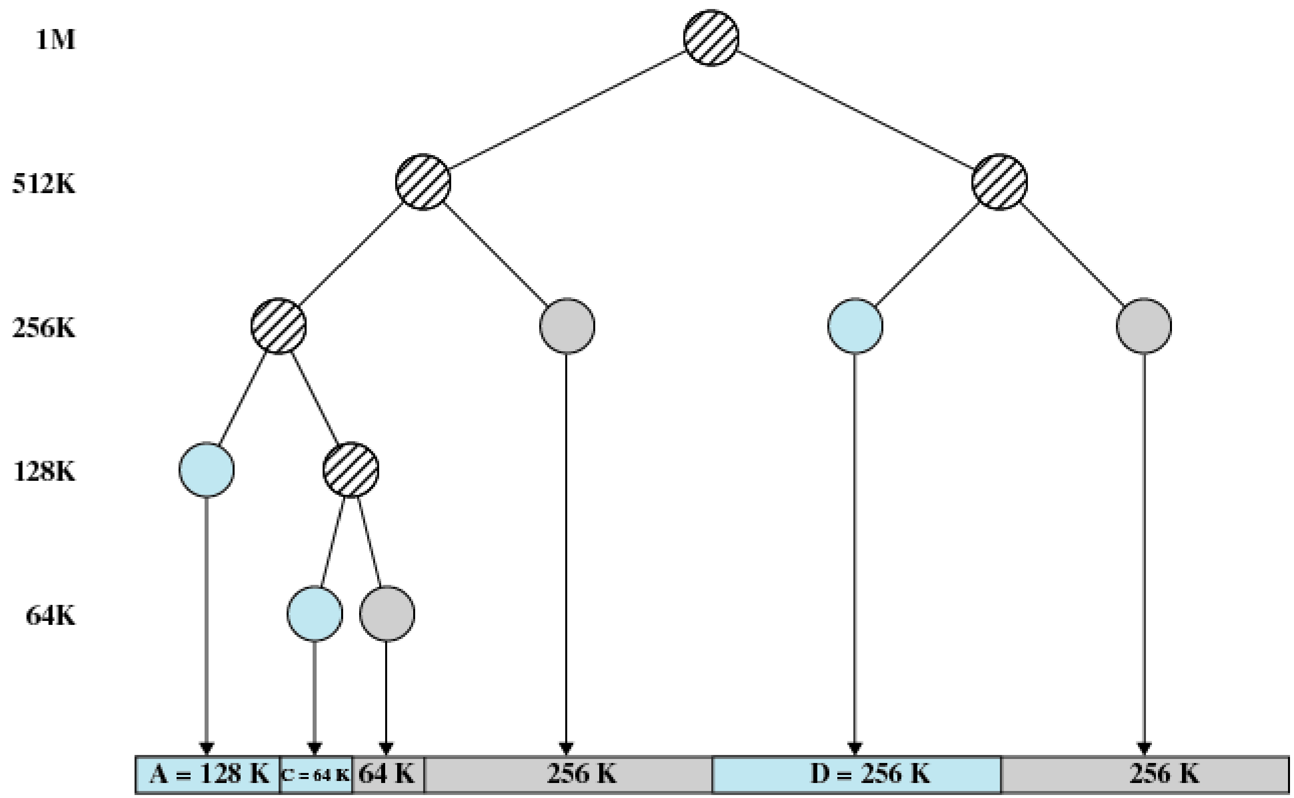


Figure 7.6 Example of Buddy System


1M
512K
256K
128K
64K
A = 128 K
128 K
256 K
512 K



Tree representation of buddy system

Relocation

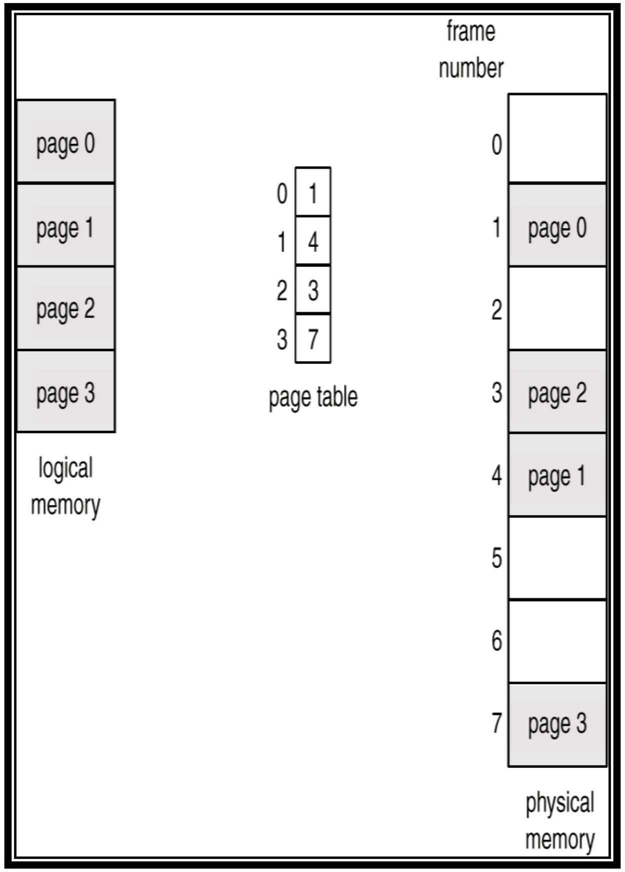
- When program loaded into memory the actual (absolute) memory locations are determined by MMU
- Without MMU it will be difficult to run multiple processes and we will soon have memory fragmentation
- A process may occupy different partitions which means different absolute memory locations during execution (from swapping)
- Compaction will also cause a program to occupy a different partition which means different absolute memory locations

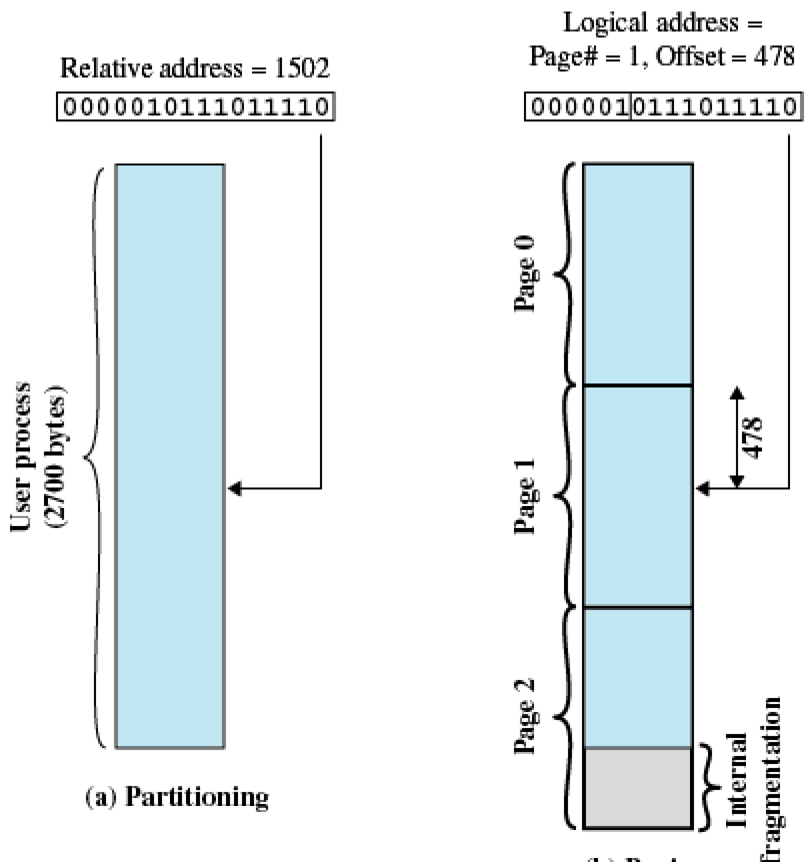
Paging

- Partition memory into small equal fixed-size chunks called pages
- Operating system maintains a page table for each process
- Memory address consist of a page number and offset within the page

Page Table Definition

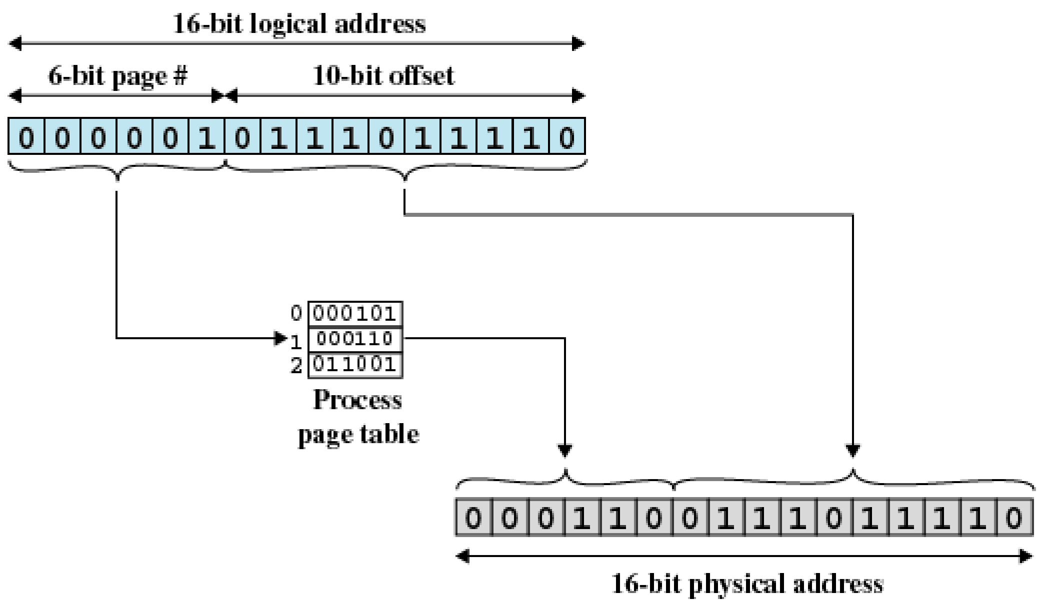
A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses





(a) Partitioning

(b) Paging
(page size = 1K)



16-bit logical address
6-bit page #
10-bit offset
0 0 0 0 0 1 0 1 1 1 0 1 1 1 1 0
0 000101
1 000110
2 011001
Process
page table
0 0 0 1 1 0 0 1 1 1 0 1 1 1 1 0
16-bit physical address

(a) Paging

Timer and Event Management

- Timer Management
  - Initialization, allocation, management and use of timers
  - Timer management library can be part of RTOS or developed independently

- Event Management
  - Involves use of event library for events like timer expiration, queuing buffers etc.

  - Event library ensures selective reception of signals by tasks
    - Select(): bit mask indicates the events that will signal a task

  - Main loop processing of events means single point of entry for all events

  - Access to SET is protected against an ISR accessing it directly

Homework

- How would you Implement a timer manager for an RTOS
- How would you implement an event manager for an RTOS

IPC

- Inter-Process Communication (IPC)
  - Communication mechanisms provided by RTOS:
    - Message Queues,
    - Semaphores for mutual exclusion,
    - Shared memory,
    - Mailboxes,
    - Signals / Events

Protocol and Driver interface

- Protocol Interfaces to multiple subsystems:
  - RTOS
  - Memory and Buffer Management
  - Timer and Event management
  - IPC
  - Device driver components
  - Configuration and control components

- Driver Interfaces
  - Device driver interface is at lowest level of OSI
  - Layering of driver for reusability and modularity
    - Device Adaptation layer → provides uniform access to higher layers (DAL)
    - Hardware Adaptation Layer (HAL)

Configuration and Control

- External Manager used for configuration, control, status and statistics
- Protocol task communicates to Manager through an agent
  - Agent is a software entity on the embedded device which processes and responds to manager requests
  - Manager-agent communication is through a standard protocol (SNMP)
  - Command Line interface is special case of manager-agent interface

Protocol Management

- Enabling and disabling the protocol
- Enabling and disabling the protocol on a specific port
- Addressing a specific interface (e.g., the IP address on a port)
- Setting maximum frame size
- Managing protocol message timeouts
- Timing out peer entities (keep-alives)
- Authenticating security information (e.g., passwords, security keys)
- Managing traffic parameters (packet loss, window size, etc)
- Encapsulation information (IP- tunnel, IPv6, IP-Sec, TCP/IP)