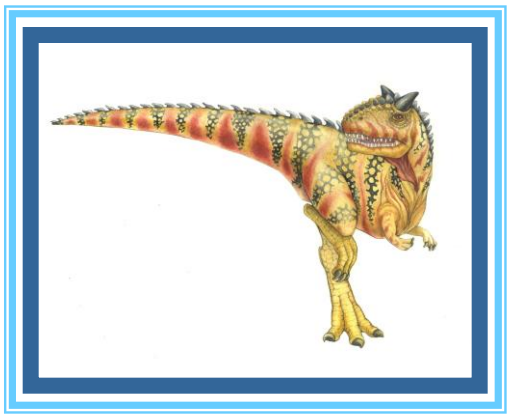
Chapter 3: Processes

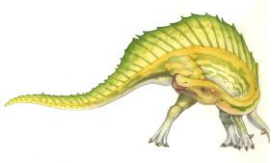


Silberschatz, Galvin and Gagne ©2018



Outline

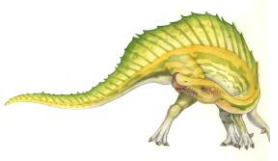
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Examples of IPC Systems
- Communication in Client-Server Systems





Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that use pipes and POSIX shared memory for interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.



Troy University Resources



Process Concept

- An operating system executes a variety of programs that run as a process.
- **A process is a program in execution. A single-threaded process has one instruction stream. Multiple threads in a process can execute in parallel on different cores.**
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time





Process Concept (Cont.)

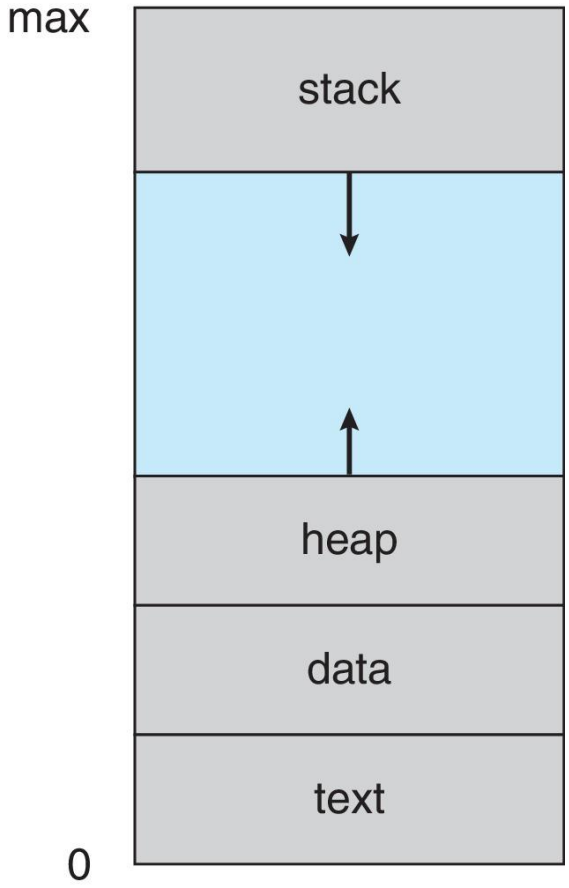
- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

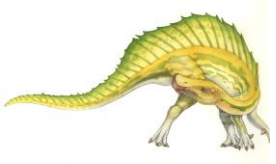


Troy University Resources



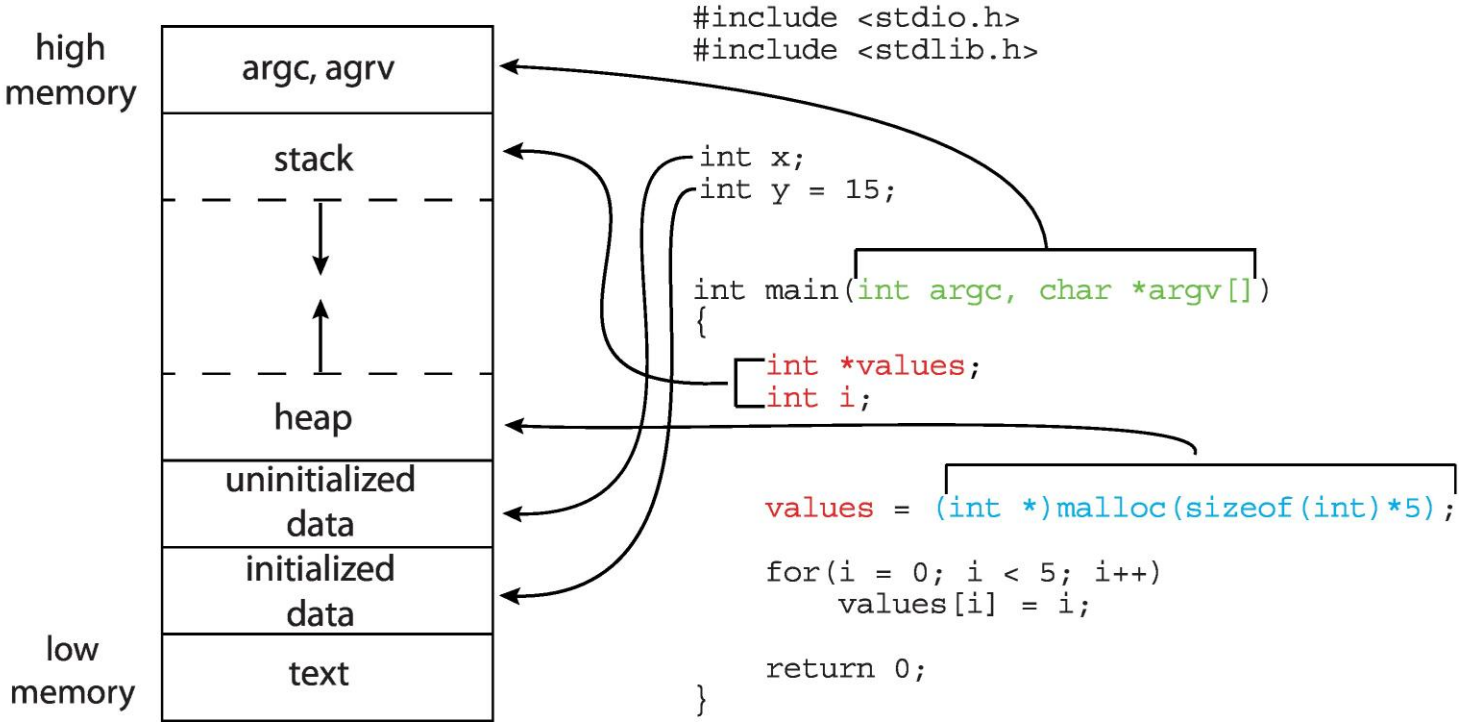
Process in Memory

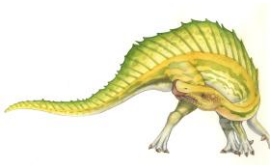






Memory Layout of a C Program





Troy University Resources



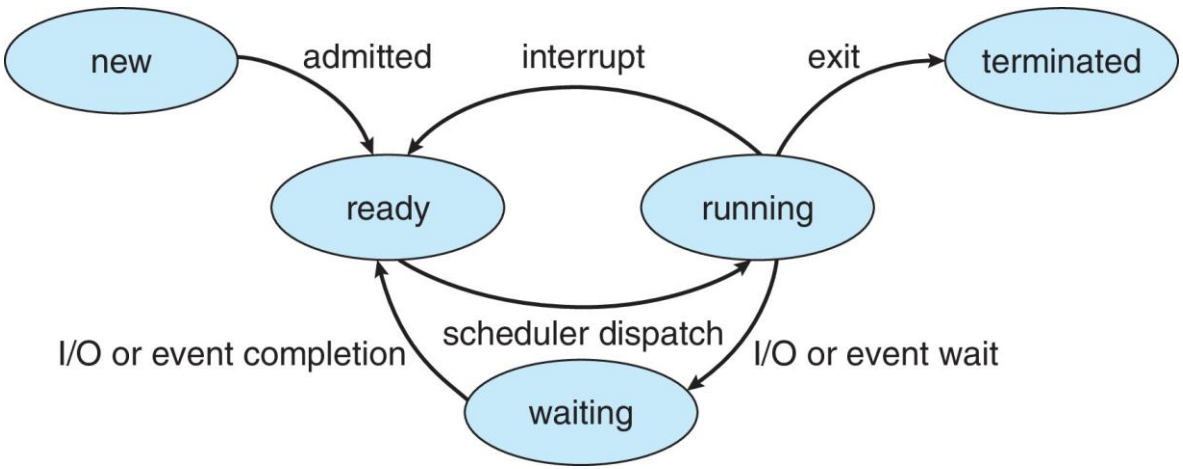
Process State

- 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 processor
  - **Terminated**: The process has finished execution





Diagram of Process State





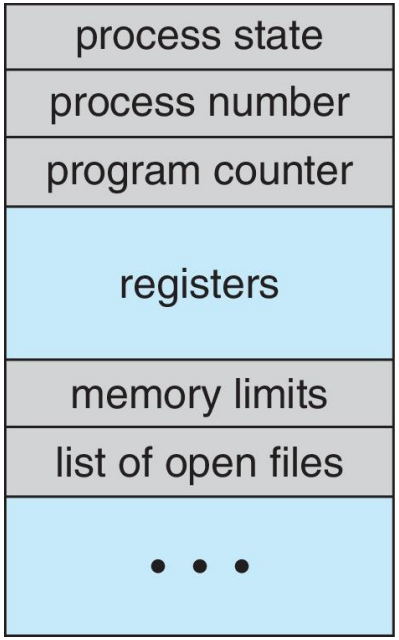
Troy University Resources



Process Control Block (PCB)

Information associated with each process(also called **task control block**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

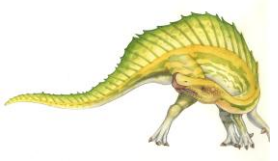






Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Explore in detail in Chapter 4



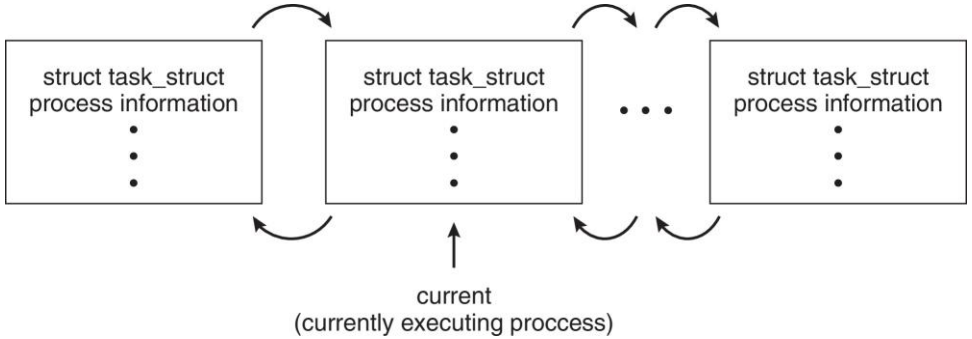
Troy University Resources



Process Representation in Linux

Linux represents each schedulable task with a task_struct. The exact fields vary with kernel version.

```
struct task_struct {
    pid_t pid;
    struct task_struct *parent;
    struct list_head children;
    struct files_struct *files;
    struct mm_struct *mm;
};
```

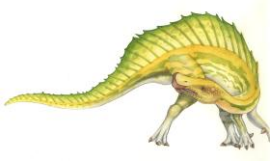






Process Scheduling

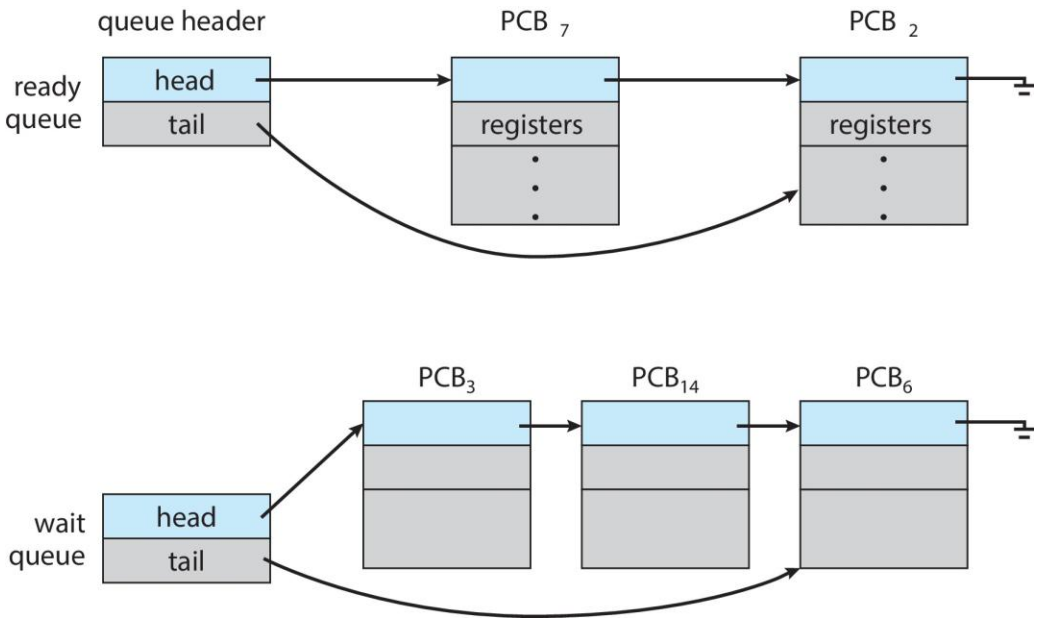
- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues



Troy University Resources



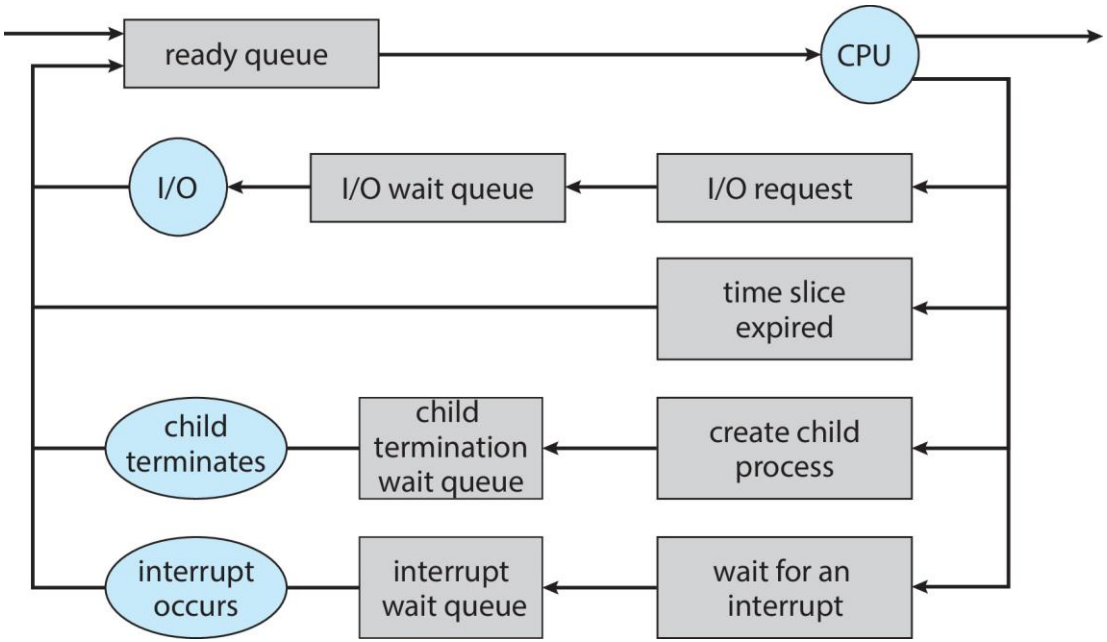
Ready and Wait Queues

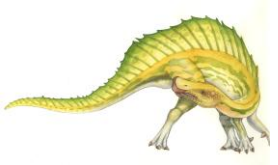






Representation of Process Scheduling





Troy University Resources



Process Schedulers and Workload Mix

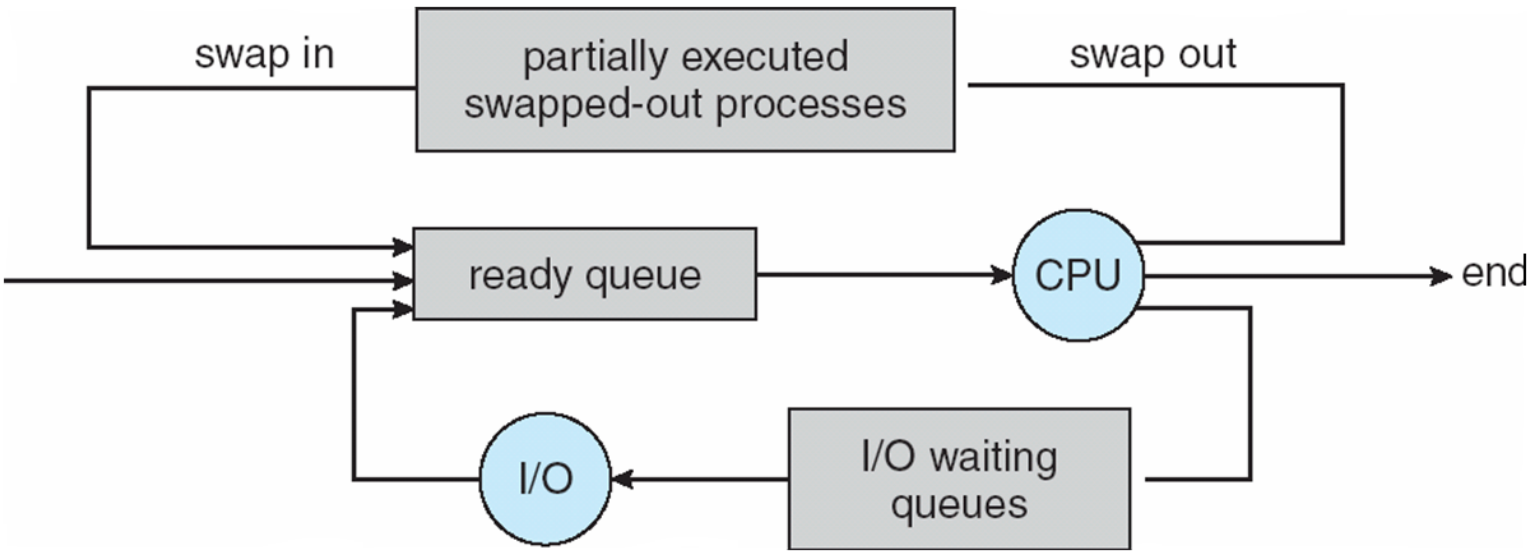
- The short-term CPU scheduler selects a ready task frequently and must run with low overhead.
- The long-term job scheduler admits work and controls the degree of multiprogramming, especially in batch systems.
- CPU-bound work tends to have longer CPU bursts.
- I/O-bound work tends to have short CPU bursts and frequent I/O waits.
- A useful mix allows computation and device activity to overlap.
- A separate long-term scheduler is not present in every general-purpose OS.





Medium-Term Scheduling

- A medium-term scheduler can suspend work to reduce memory pressure or multiprogramming.
- In classic swapping, a process leaves RAM and is later restored to continue.



Modern systems often reclaim individual pages. A suspended process is distinct from a ready process waiting for CPU time.

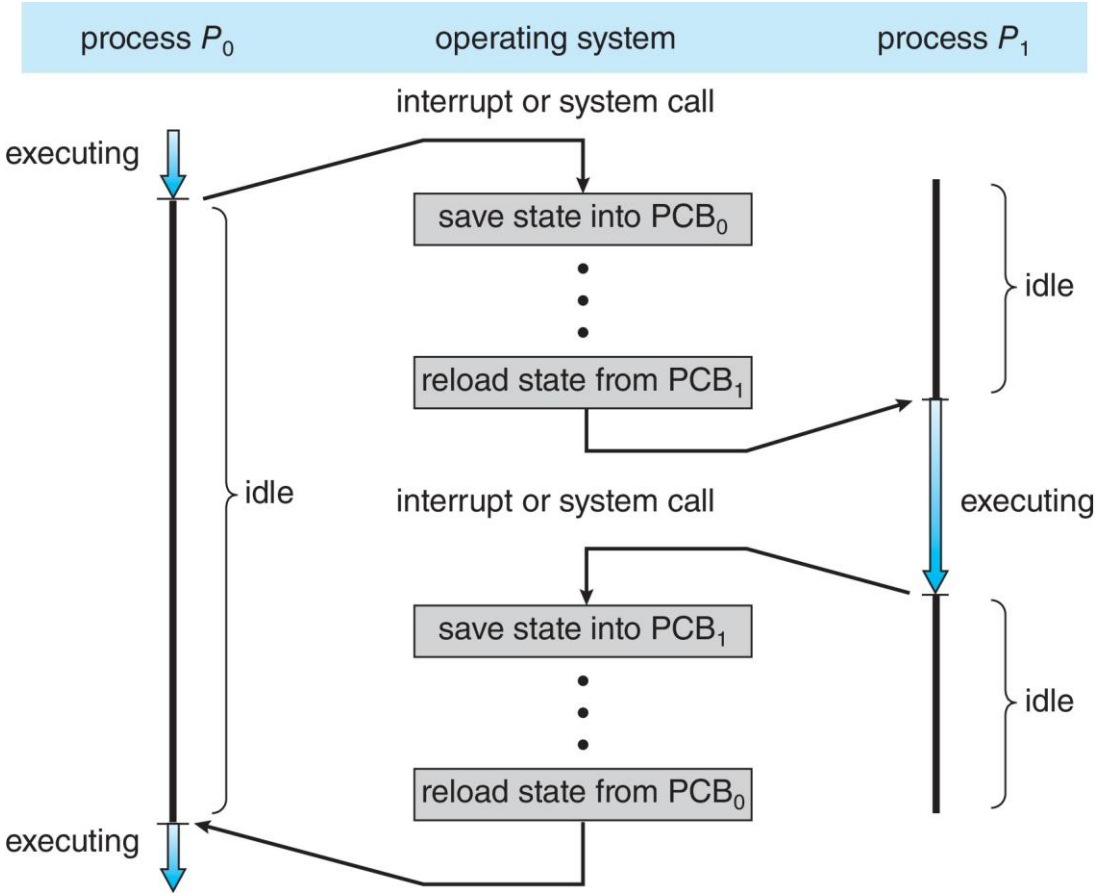


Troy University Resources



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

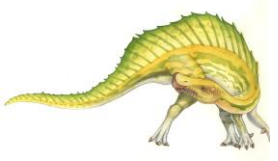






Context Switch

- 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 via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once



Troy University Resources



Multitasking in Mobile Systems

- Mobile systems schedule many processes but restrict background work to save power and memory.
- Foreground and visible applications receive priority over less important work.
- Background execution must use the platform's supported services, jobs or task APIs.
- A service does not guarantee that an Android process will run indefinitely.
- Early iOS restrictions are historical examples. Current limits depend on OS version and the kind of task.





Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination



Troy University Resources



Process Creation

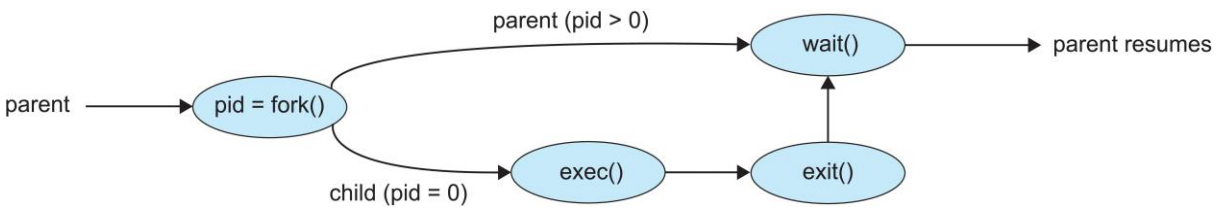
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

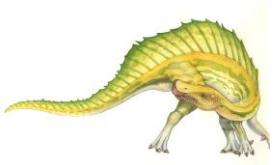




Process Creation APIs

- fork() creates a child. It returns 0 in the child, the child PID in the parent, and -1 on failure.
- A successful exec call replaces the process image and does not return. A child must handle a failed exec and exit appropriately.
- wait() or waitpid() lets the parent collect a terminated child's status.
- For CreateProcessW(), supply a writable command-line buffer. Initialize the startup structures, check success, and close the returned handles.

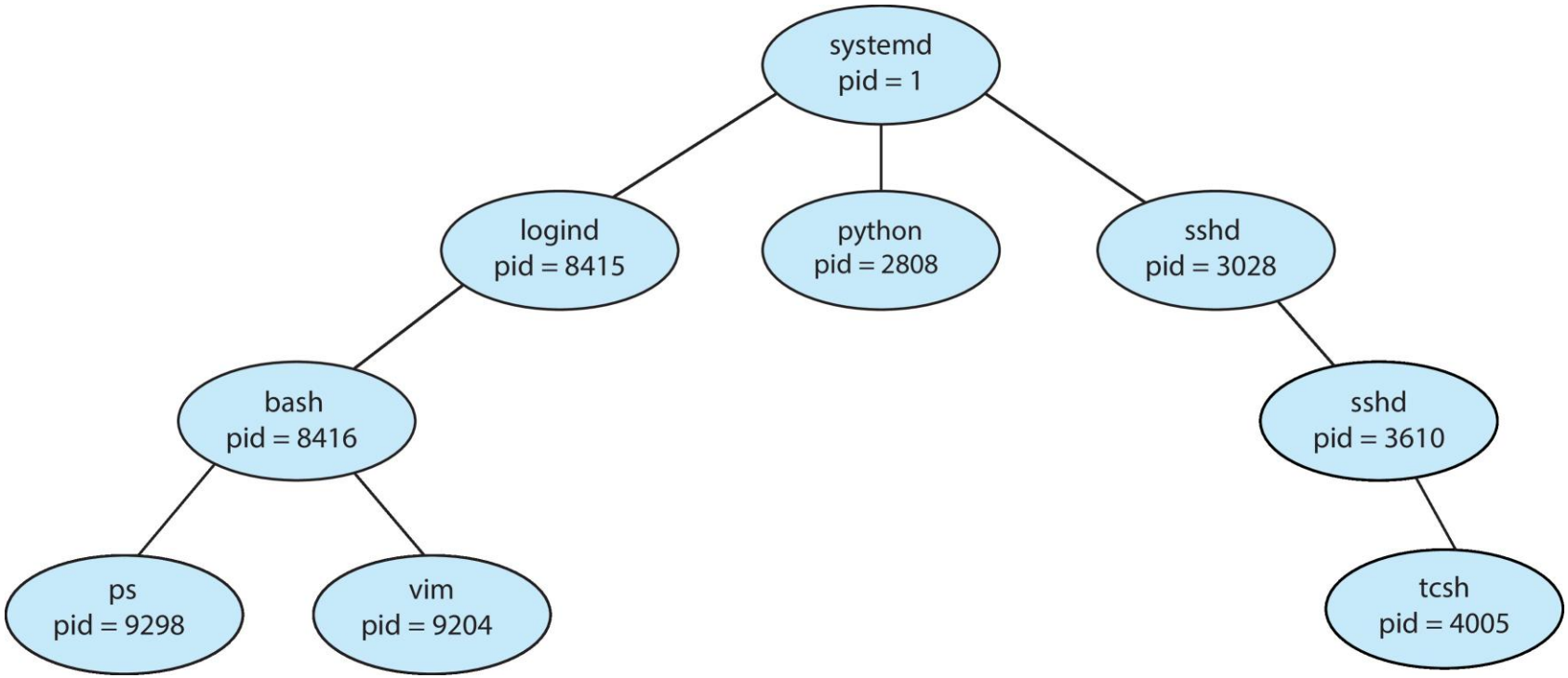


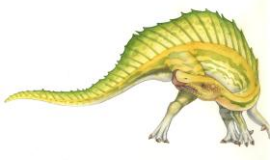


Troy University Resources



A Tree of Processes in Linux







C Program Forking Separate Process

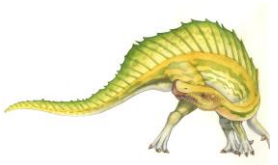
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```





Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

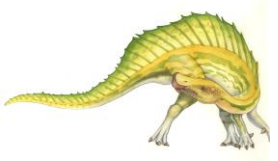
    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```





Process Termination

- A process can terminate by returning from main(), calling exit(), or receiving a terminating signal.
- The kernel releases most process resources and retains termination status until it is collected.
- The parent uses wait() or waitpid() to obtain the child's status.
- On POSIX systems, a parent can request termination with kill(child_pid, SIGTERM), subject to permissions.
- abort() terminates the calling process abnormally. It does not take a child PID.
- Operating systems may also terminate tasks for policy or resource reasons.



Troy University Resources



Process Termination

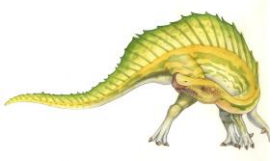
- A zombie is a terminated child whose status has not yet been reaped.
- It retains a small bookkeeping record, rather than continuing to execute.
- An orphan is a process whose parent has terminated. It may continue to run.
- Linux reparents orphans to a suitable subreaper or init process, which can reap them.
- wait(&status) returns the PID of a terminated child and reports how it ended.
- Cascading termination is a policy of some systems. Parent exit does not universally terminate children.





Android Process Importance Hierarchy

- Android ranks processes by their importance to the user.
- Foreground processes perform work essential to the current interaction.
- Visible processes contribute to something the user can still see.
- Service processes perform supported ongoing work.
- Cached processes can be killed to reclaim memory. Exact categories and restrictions depend on Android version.
- The older background/empty split is a textbook-era description of lower-priority cached work.

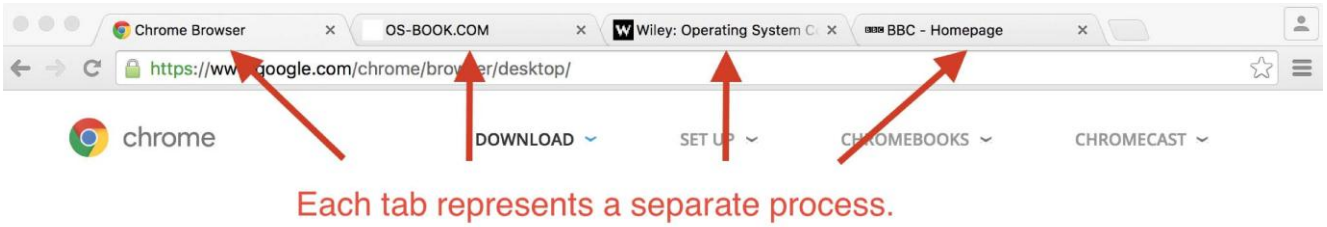


Troy University Resources



Multiprocess Architecture – Chrome Browser

- Chromium separates the browser interface from sandboxed renderer processes.
- Site Isolation can separate content within one tab across processes. There is not necessarily one renderer for every tab.
- GPU, network and other utility processes handle additional functions.
- The figure shows a simplified historical model of process assignment.
- Process separation helps contain renderer failures and enforce security boundaries.

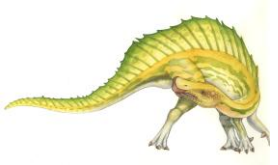






Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**



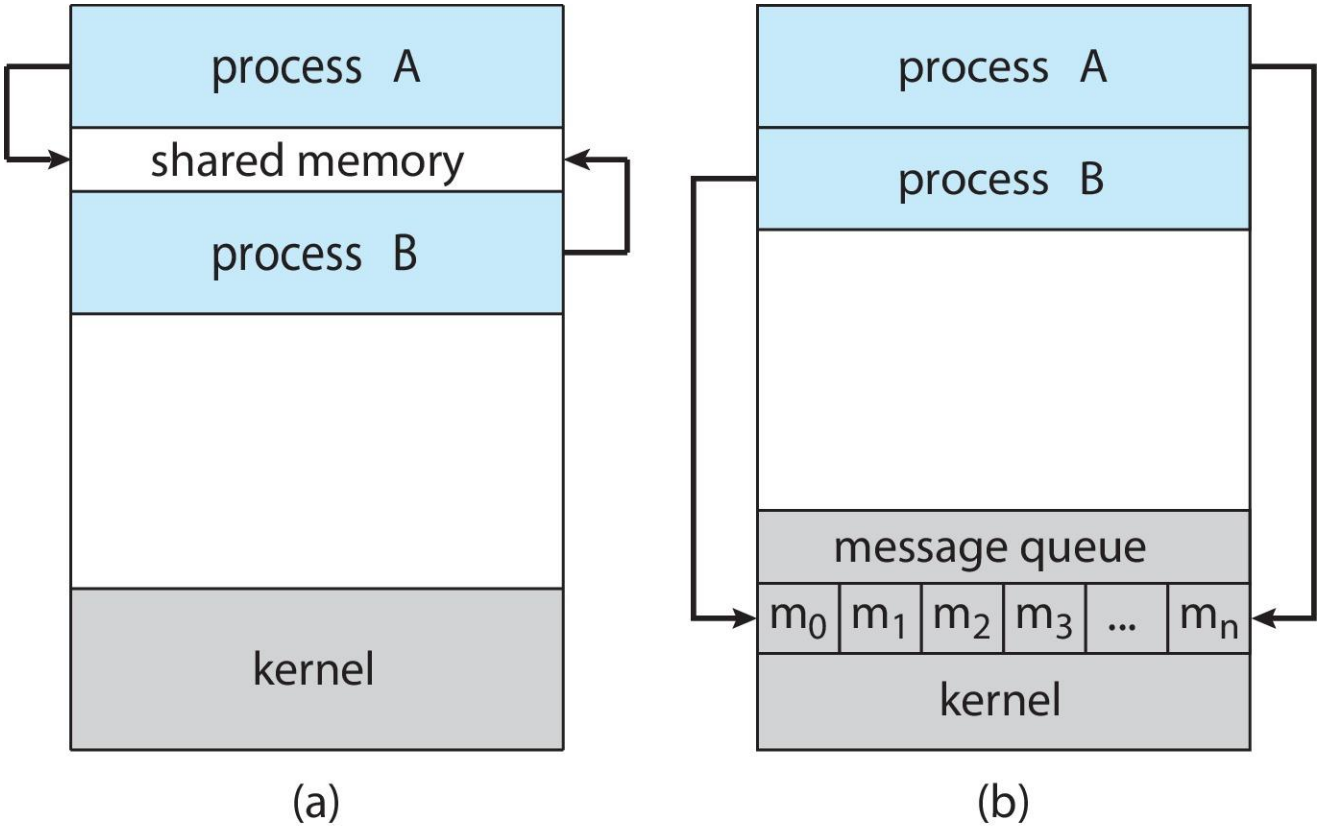
Troy University Resources



Communications Models

(a) Shared memory.

(b) Message passing.







Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

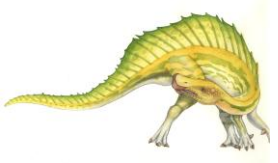


Troy University Resources



IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.





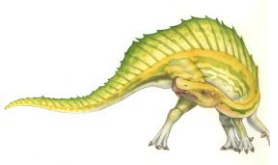
Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The single-producer/single-consumer algorithm uses BUFFER_SIZE - 1 slots, assuming atomic index access and correct memory ordering.





Producer Process – Shared Memory

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```



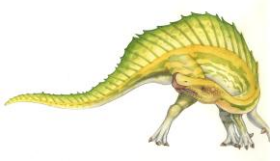


Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

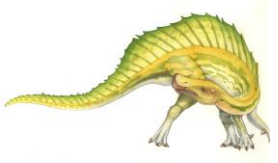


Troy University Resources



What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The consumer decrements the counter after it consumes an item.





Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

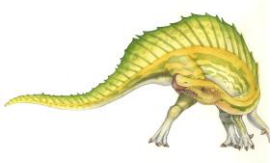


Troy University Resources



Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```





Race Condition

- **`counter++`** could be implemented as

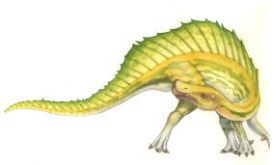
```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}



Troy University Resources



Race Condition (Cont.)

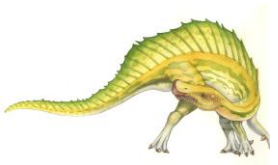
- In the first algorithm, only the producer updates in and only the consumer updates out.
- The counter-based variant adds a shared read-modify-write variable that both processes update.
- counter++ and counter-- can interleave and lose an update.
- The first algorithm still assumes one producer, one consumer, atomic index access and suitable memory ordering.
- Ordinary shared C variables alone do not provide a portable concurrent implementation. Atomics or synchronization are needed.





IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable



Troy University Resources



Message Passing (Cont.)

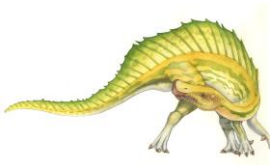
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?





Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

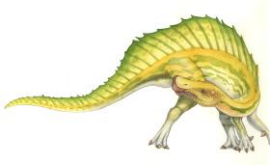


Troy University Resources



Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional





Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional



Troy University Resources



Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A





Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



Troy University Resources



Synchronization

- A blocking receive waits until a message is available.
- A blocking send waits until the API's completion condition is satisfied, for example acceptance into a buffer or receipt by the receiver.
- A nonblocking operation returns promptly and reports success, lack of data or lack of space according to the API.
- A rendezvous requires sender and receiver to meet, as in a zero-capacity channel.
- Blocking send and receive alone do not imply a rendezvous when the channel buffers messages.





Producer-Consumer: Message Passing

Abstract blocking channel operations. send() waits when the channel cannot accept another item.

```
/* Producer */
while (true) {
    message item = produce();
    send(channel, item);
}

/* Consumer */
while (true) {
    message item = receive(channel);
    consume(item);
}
```

These are pseudocode operations. A real API also defines errors, buffering and shutdown behavior.

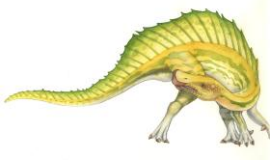


Troy University Resources



Buffering

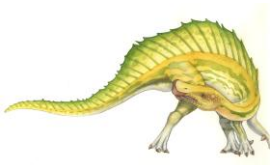
- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits





POSIX Shared Memory

- shm_open() creates or opens an object. Use the same portable name, such as "/OS", in both programs.
- Set its size with ftruncate() before mmap(). Include <sys/mman.h> and <unistd.h> for these interfaces.
- Check descriptors, sizes and return values, including MAP_FAILED, before accessing the mapping.
- Use char * for portable byte-wise pointer arithmetic. Keep writes within the mapping and terminate strings before printing them.
- A shared mapping does not synchronize access. For this example, complete the producer before the consumer reads.
- munmap() removes a mapping, close() closes a descriptor, and shm_unlink() removes the object's name.



Troy University Resources



IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```





IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```



Troy University Resources



Examples of IPC Systems - Mach

- Mach IPC communicates through ports. Port rights control which tasks may send or receive messages.
- Many Mach services use message-based RPC. Some kernel interfaces are direct traps.
- mach_msg() sends a message, receives a message, or performs both operations.
- mach_port_allocate() can create a receive right. Establishing send rights is a separate step.
- A full port queue can block a sender. Send and receive timeout options control waiting.
- Check the return status. The following slides show message-transfer excerpts.





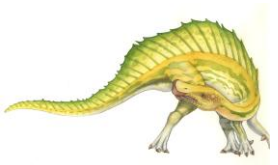
Mach Messages

```
#include <mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;
```

The next slides show message-transfer excerpts. A complete program must allocate receive rights, establish send rights and initialize the header fields.



Troy University Resources



Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
   MACH_SEND_MSG, // sending a message
   sizeof(message), // size of message sent
   0, // maximum size of received message - unnecessary
   MACH_PORT_NULL, // name of receive port - unnecessary
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

Set msgh_bits to match the supplied port rights, initialize the payload, and check the return code.





Mach Message Passing - Server

```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
   MACH_RCV_MSG, // sending a message
   0, // size of message sent
   sizeof(message), // maximum size of received message
   server, // name of receive port
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```

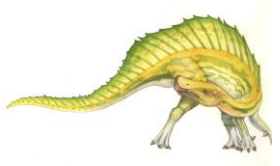
Reserve receive-buffer space for the Mach message trailer and set the receive size accordingly. Read the payload only after KERN_SUCCESS.





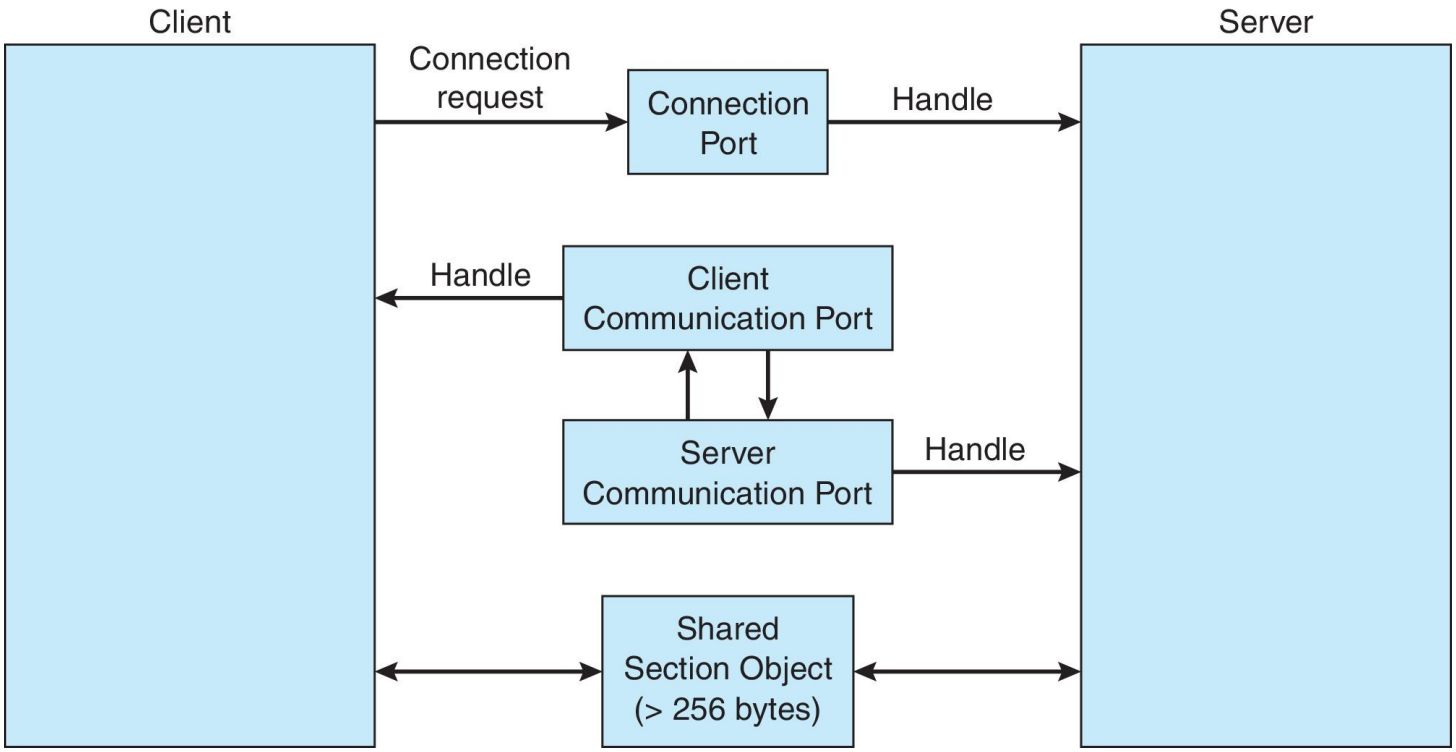
Examples of IPC Systems – Windows

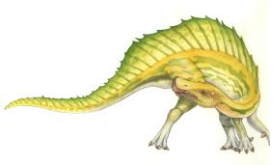
- Modern Windows uses Advanced Local Procedure Call (ALPC), the successor to LPC, for local IPC
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.





Local Procedure Calls in Windows





Troy University Resources



Pipes

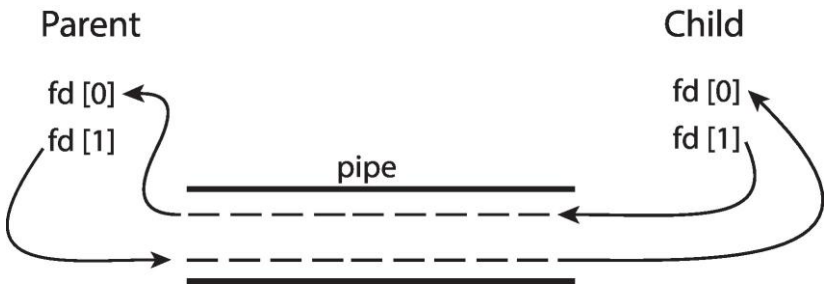
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **An ordinary pipe is accessed through open handles or descriptors, often inherited after fork(). Unrelated processes can also communicate if they obtain the handles or descriptors.**
- **Named pipes** – can be accessed without a parent-child relationship.





Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Parent and child commonly share inherited pipe descriptors. A parent-child relationship is not a fundamental requirement.



- Windows calls these **anonymous pipes**



Troy University Resources



Named Pipes

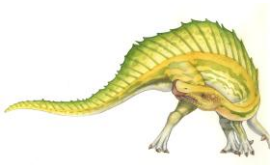
- A named pipe has a name through which unrelated processes can open the same IPC channel.
- POSIX FIFOs provide a local byte stream with ordinary pipe semantics.
- Portable bidirectional POSIX communication normally uses two FIFOs or a socket.
- Windows named pipes can be one-way or duplex and can support local or network communication.
- Multiple processes may open a named pipe, subject to the API's synchronization and connection rules.





Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls



Troy University Resources



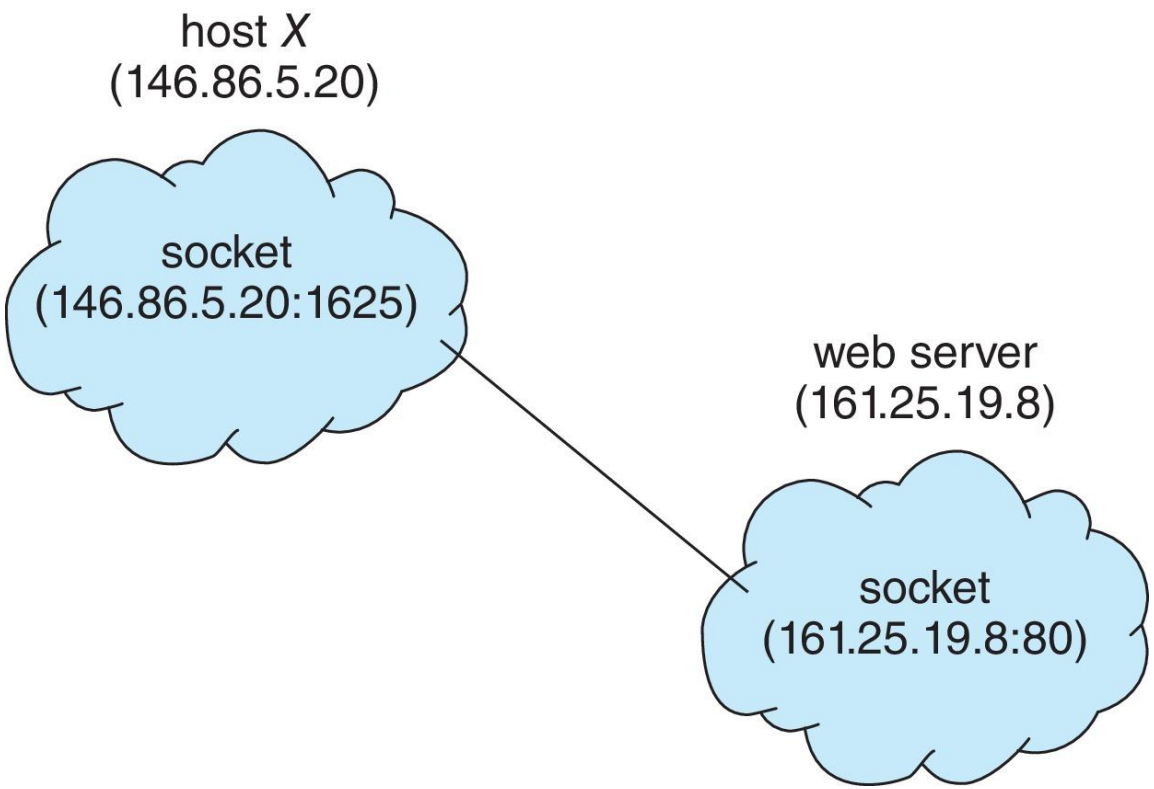
Sockets

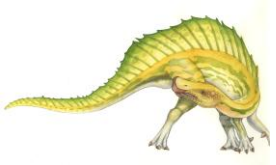
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- Ports 0–1023 are the system or well-known port range. Individual assignments are defined by IANA.
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





Socket Communication





Troy University Resources



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```





Sockets in Java

The equivalent Date client

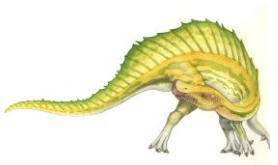
```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```





Remote Procedure Calls

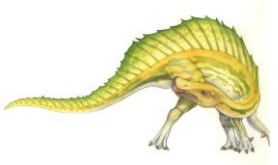
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)





Remote Procedure Calls (Cont.)

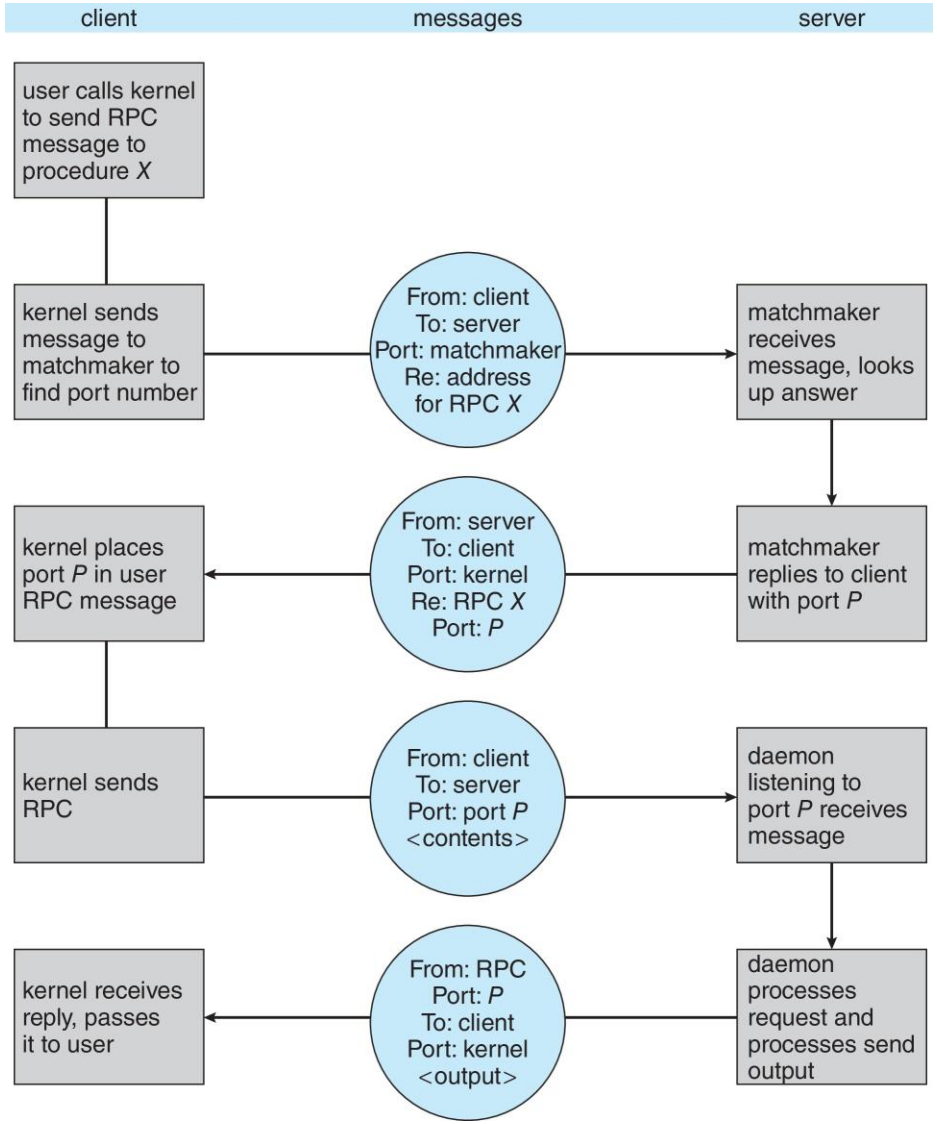
- RPC protocols need an agreed data representation. ONC RPC uses External Data Representation (XDR).
- Serialization handles differences such as byte order and data layout.
- A timeout can leave the caller unsure whether the server executed the operation.
- Retries can duplicate an operation unless the protocol or application detects duplicates.
- At-most-once or stronger behavior requires explicit protocol and failure-recovery support.
- A binder, registry or known endpoint can help clients locate a service.

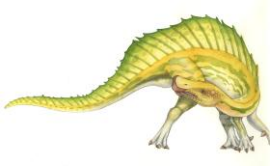


Troy University Resources

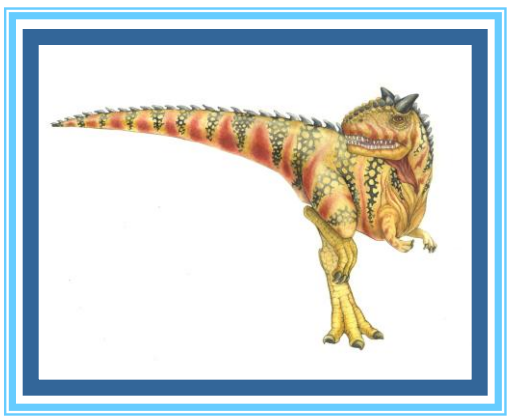


Execution of RPC





End of Chapter 3



Silberschatz, Galvin and Gagne ©2018