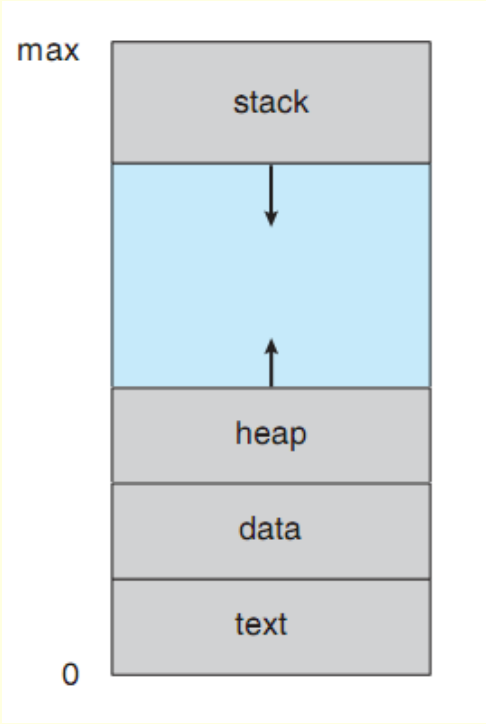
Part Two: Process Management

Processes-Threads-Synchronization-Scheduling-Deadlock

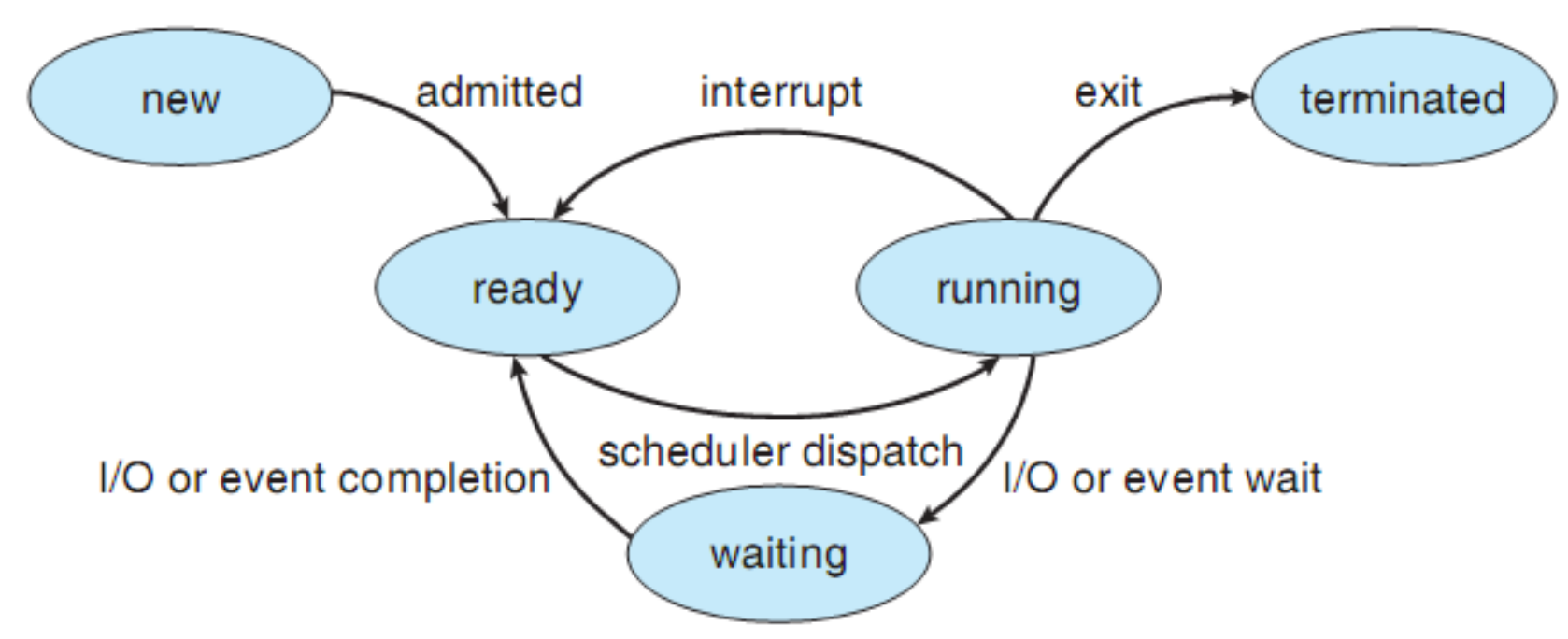
Mehdi Kargahi
School of ECE
University of Tehran
Summer 2016

Processes

- *Program*: A passive entity
- *Process*: An active entity (a program that is loaded into memory with program counter, resources)
- Single-thread vs. multiple-thread processes



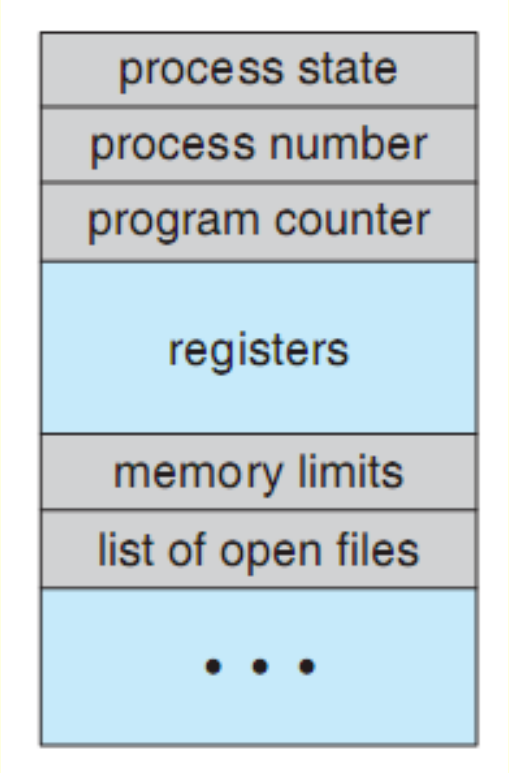
Process States



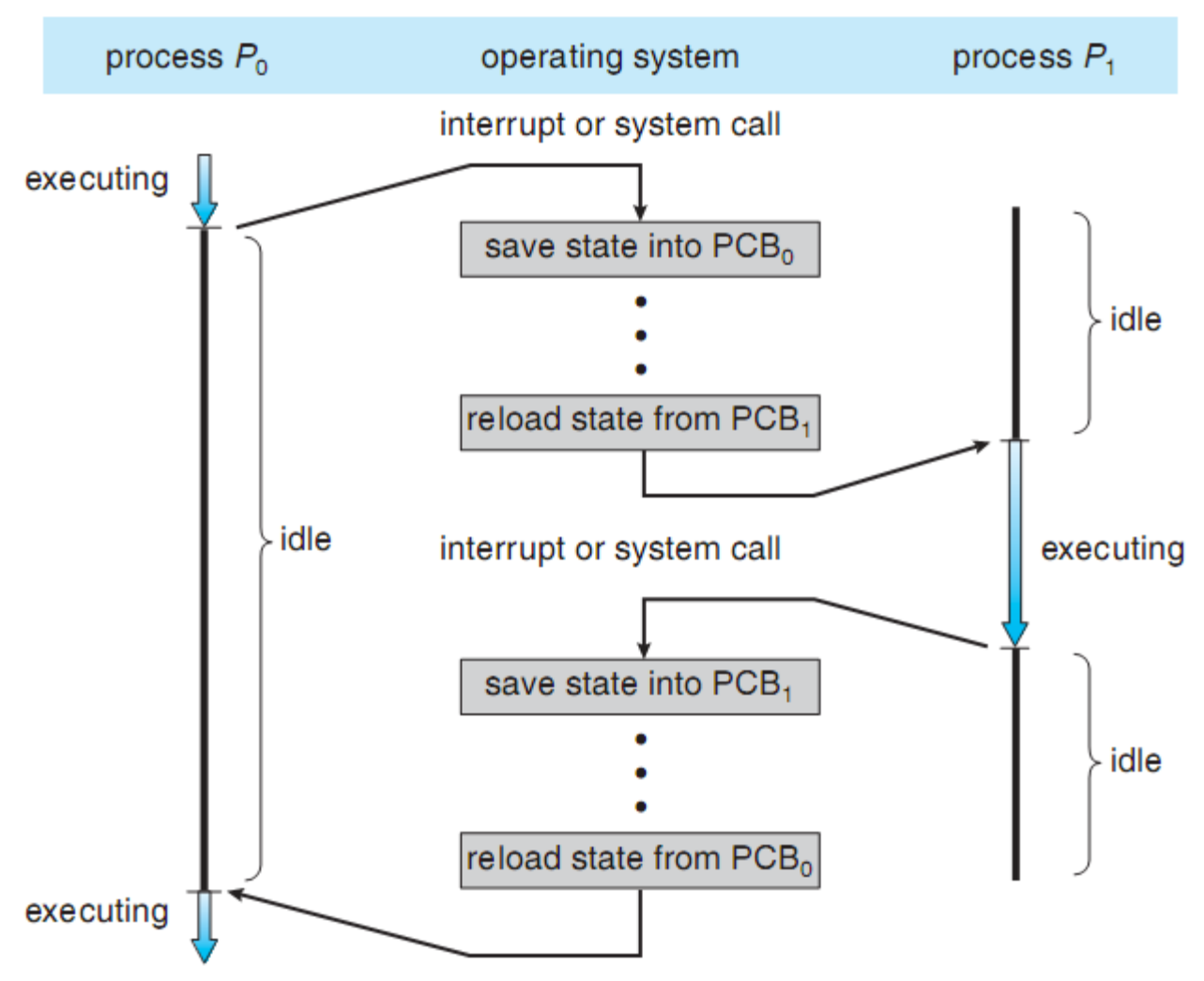
Process Control Block (PCB)

- Process state
- Program counter
- CPU registers
- CPU-scheduling information
- Memory management information
- Accounting information
- I/O status information

Process Control Block (PCB)



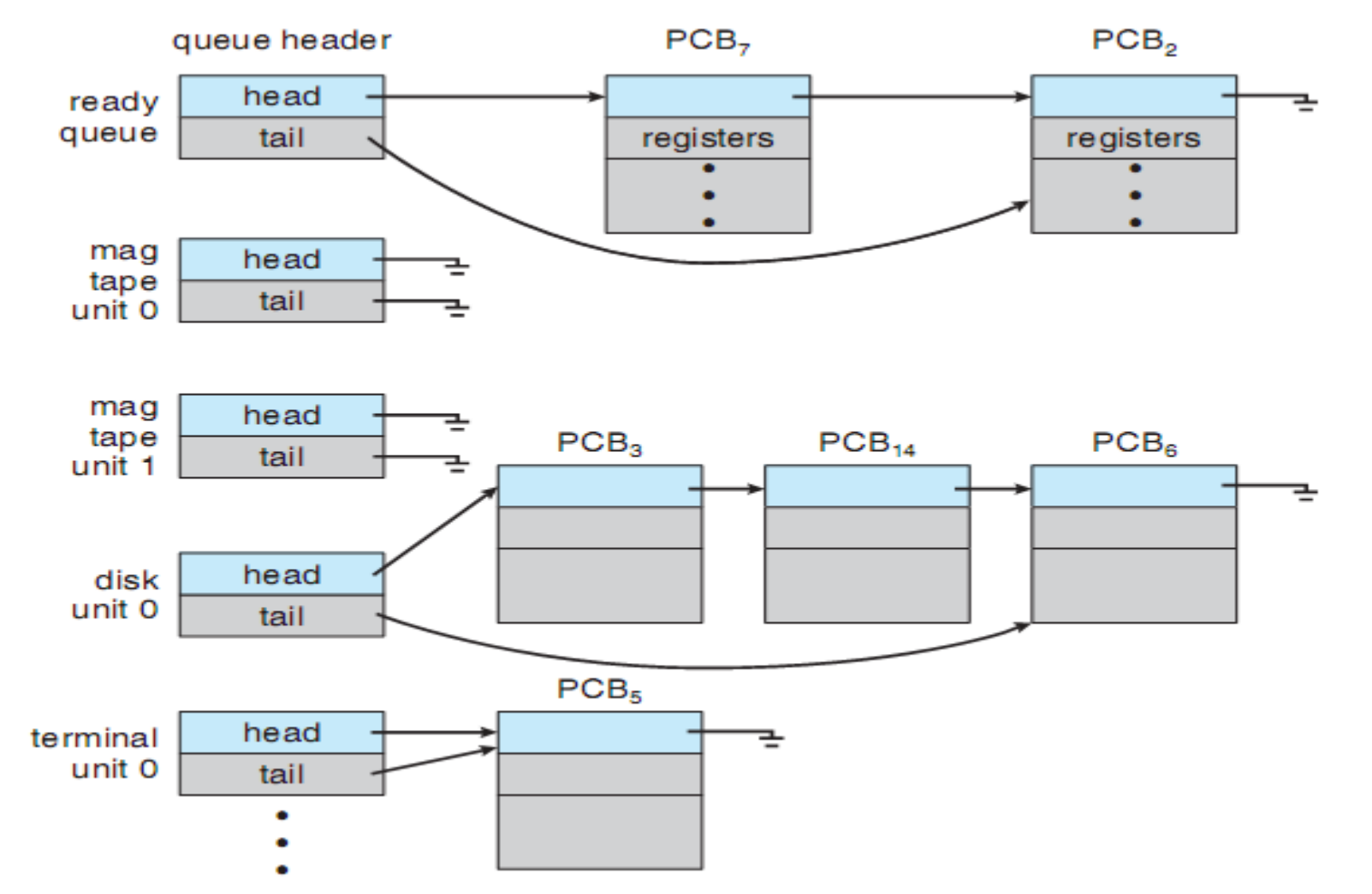
Context Switch



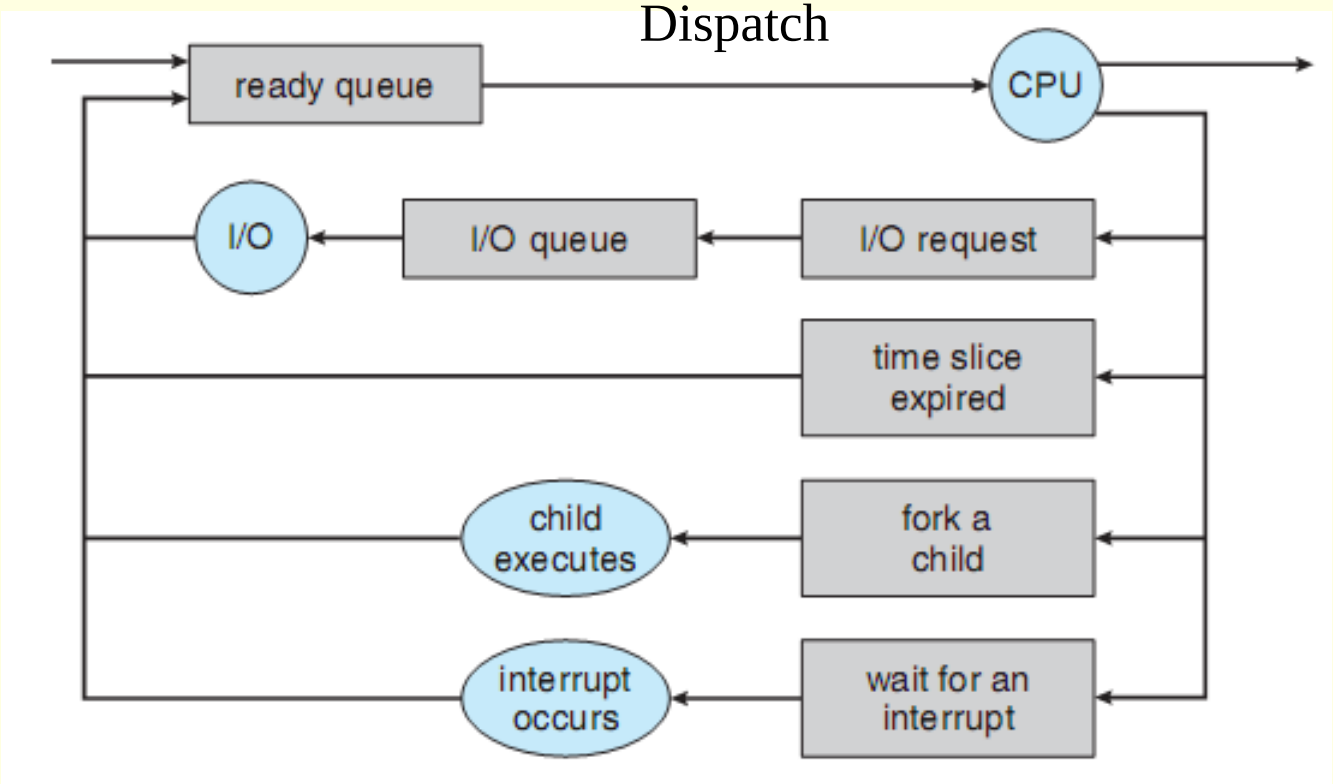
Process Scheduling

- Context switch
  - Multiple register sets for faster switching in some processors (such as Sun UltraSPARC)
- Job queue
- Ready queue (as a linked list)
- I/O devices queue (linked lists)

Ready Queue and I/O Device Queues



Queueing-Diagram Representation of Process Scheduling

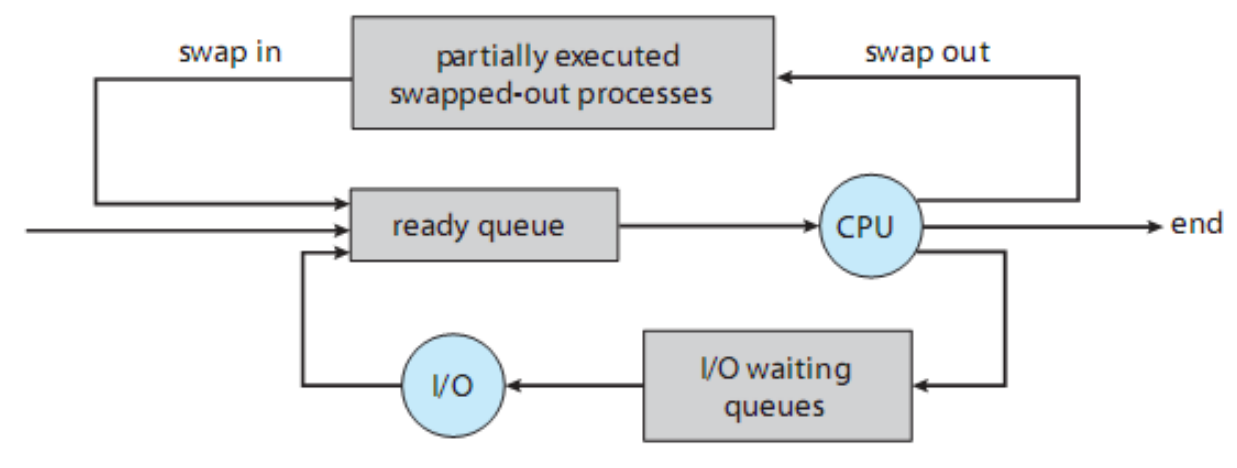


Schedulers

- *Long-term scheduler* or job scheduler
  - Often in a batch system, from the spooled requests
  - Controls the *degree of multiprogramming*
  - Is not present in all operating systems
  - Should prepare a good mix of I/O-bound and CPU-bound processes
  - Stability: process creation rate ≤ process completion rate
- *Short-term scheduler* or CPU scheduler
  - High frequency (e.g., every 10^{ms} to 100^{ms})
  - Context switch overhead should be kept low

Schedulers

- Some operating systems introduce an intermediate level of scheduling
- *Medium-term scheduler* to dynamically changing the degree of multiprogramming using *swapping*
- Added to the previous queueing diagram



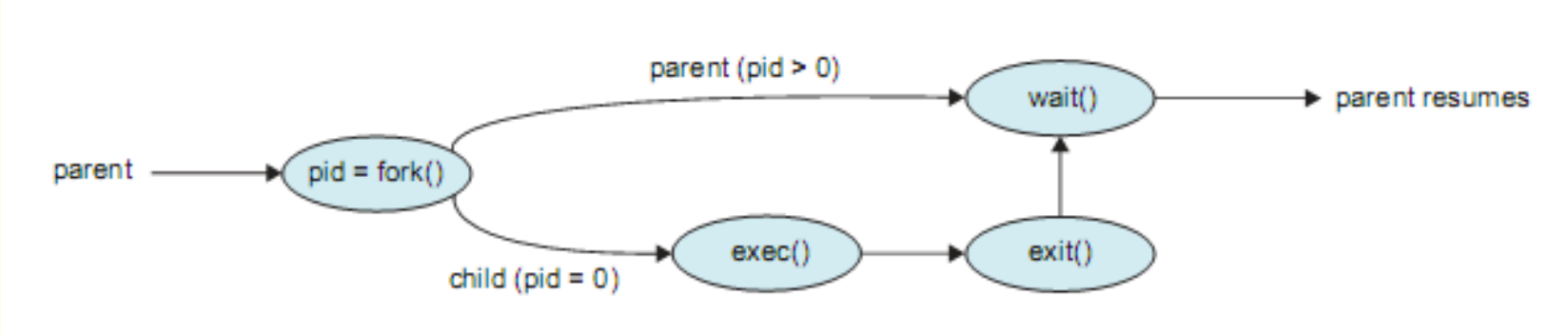
Operations on Processes

- Process creation
  - A *parent* process may create several new *children* processes via specific *system calls*
  - *Pid:* Process identifier
  - Each process is created by its parent (except the first process)
- Possibilities after creating a process
  - The parent continues execution concurrently with its child
  - The parent waits until some or all of its childs have terminated
- Possibilities in terms of the address space of the new process
  - The child is a duplicate of the parent (same program & data)
  - The child process has a new program loaded into it

UNIX Fork System Call

```
#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;
}
```

UNIX Fork System Call



Creating a Separate Process using Win32 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);
}
```

Operations on Processes

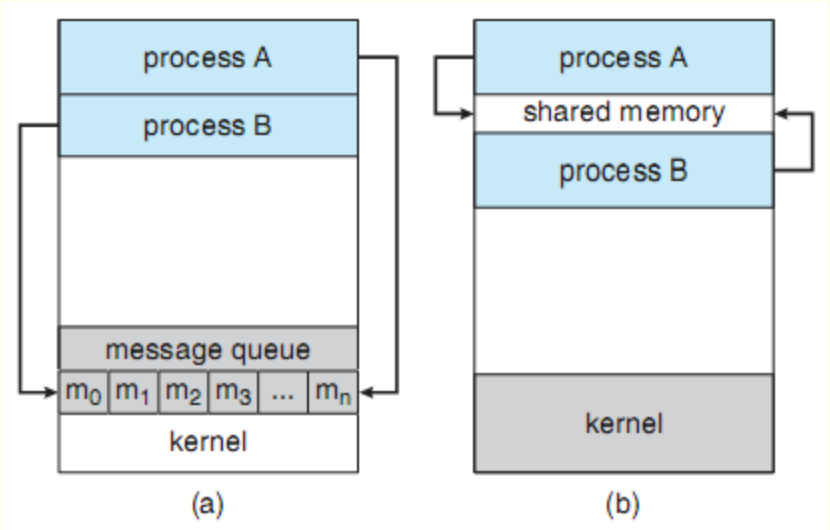
- Process termination
  - Calling exit() or aborted by another process (normally its parent)
  - Cascading termination
- The reasons that a parent may terminate its children
  - The child has exceeded its usage of some of its resources
  - The child task is no longer required
  - The parent is exiting (depends on the OS)
    - VMS (cascading termination)
    - UNIX (re-parent it)

Process Communication

- Independent processes
  - No shared data
  - Deterministic and repeatable execution
- Cooperative processes
  - Shared data (normally through shared memory)
  - Mutual effect on each other
  - Non-deterministic (stochastic) execution

Example

- Producer-Consumer
  - Shared memory (b)
  - Message passing (a)
    - Bounded buffer
    - Unbounded buffer
- Message passing
  - Direct communication
  - Indirect communication (mailboxes or ports)



Example

- Message passing
  - Blocking (Synchronous)
  - Non-blocking (Asynchronous)
- Blocking send
- Non-blocking send
- Blocking receive
- Non-blocking receive (returns a valid message or a null)
- Buffering
  - Zero capacity
  - Bounded capacity
  - Unbounded capacity

Examples of IPC Systems

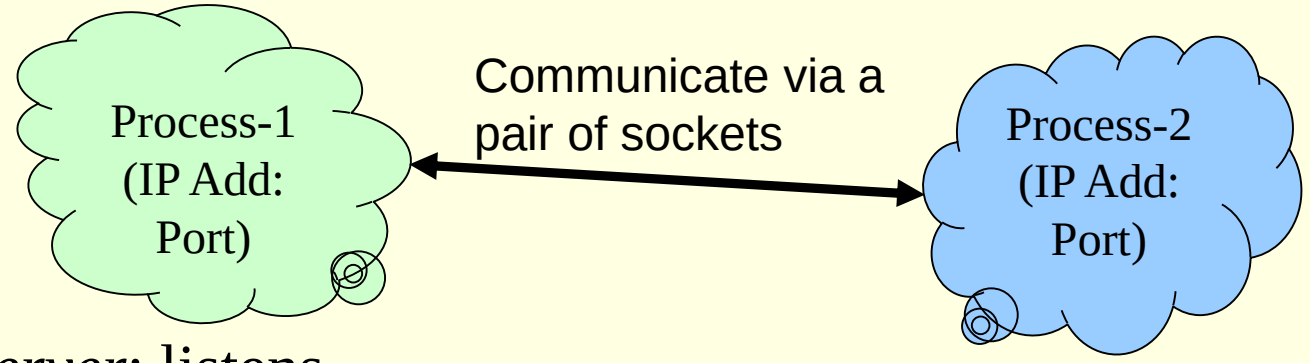
- POSIX shared memory
  - Create a shared memory segment
    - Segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR)
  - Attach it to the process address space
    - Shared_memory = (char *)shmat(id, null, 0)
  - Accessing the shared memory as a routine memory
    - Sprintf(shared_memory, “Writing to shared memory”)
  - When no longer is required: detach the segment
    - Shmdt(shared_memory);

Other Strategies for Communication in C/S Systems

- Sockets
- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)
- Pipes

Sockets

- *IP address* specifies the host
- *Port* specifies the communicating process

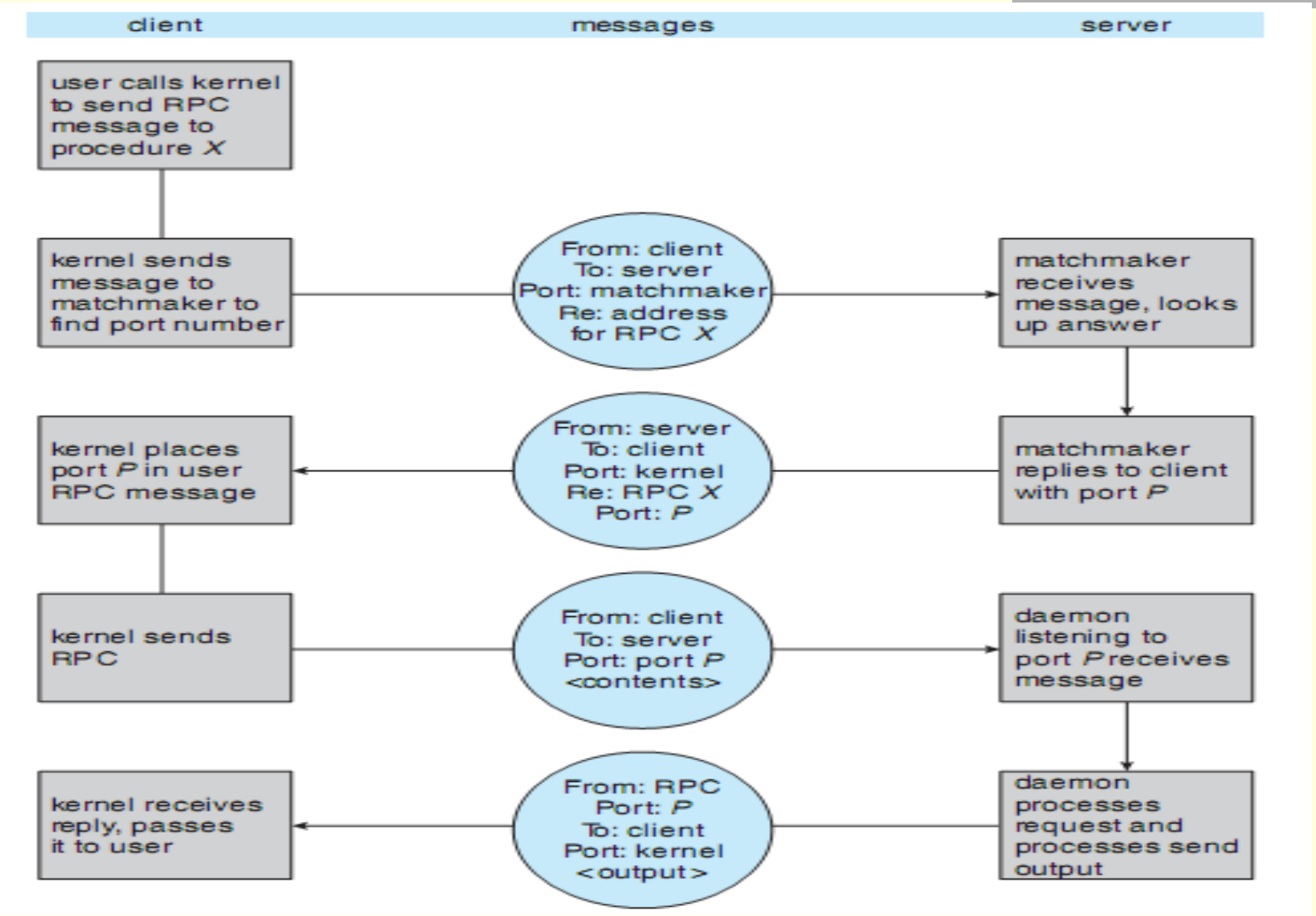


- *Server*: listens
- *Client*: creates a socket and requests a connection to the server as *IP: Port*
- *Property*: unstructured stream of bytes. C & S should know their structures

RPC (Remote Procedure Call)

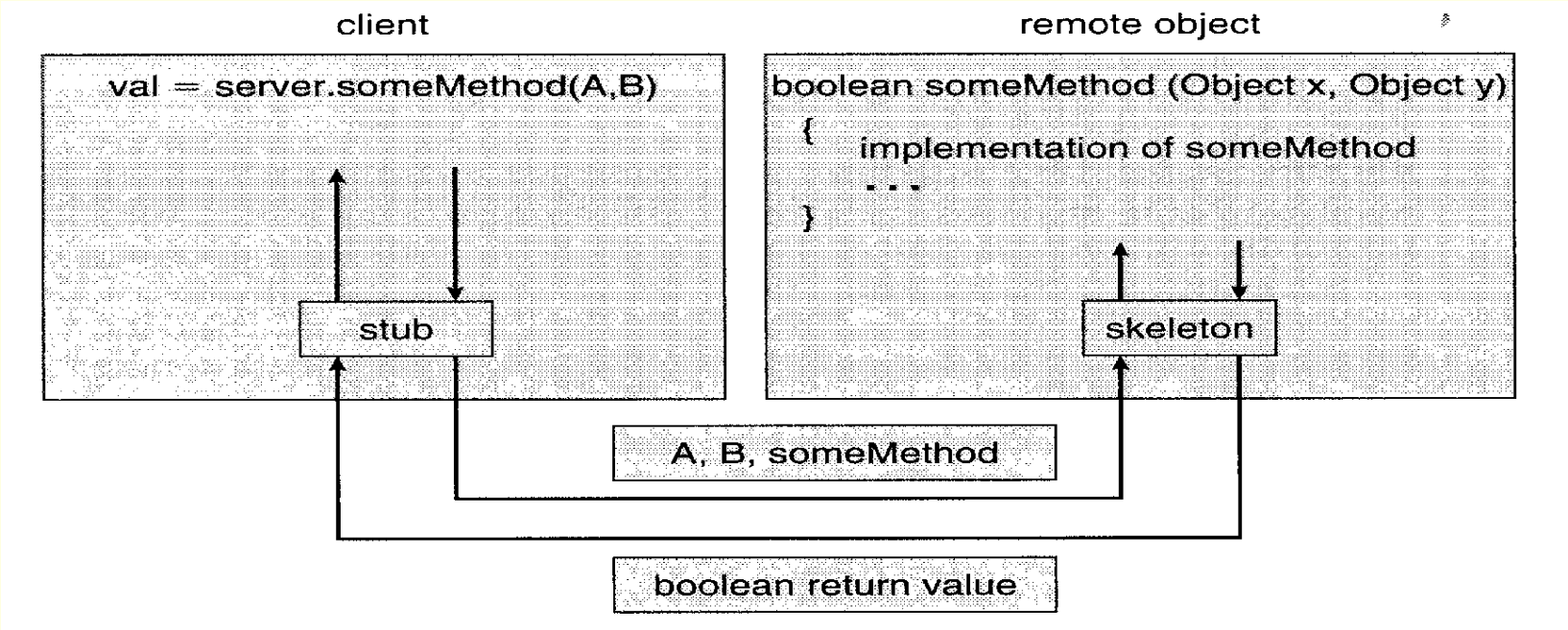
- Structured byte stream as the required information for initiating and doing a procedure call
- An RPC daemon which is listening to a port on a remote machine may get a message containing
  - Identifier of the function to be executed
  - The parameters which are passed to the function (*marshaled*)
- The output is sent back in a separate message
- Finding the port of a service in RPC
  - Predetermined binding information (fixed port address at compile time)- *less overhead*
  - Dynamic binding (using rendezvous/matchmaker daemons)- *more flexibility*

RPC (Remote Procedure Call)



RMI (Remote Method Invocation)

- A Java feature (invoking a method on a remote object)
- Objects are remote if they are on a different JVM
- Client stub marshals the method-name and parameters
- Skeleton on the server un-marshals them



RMI (Remote Method Invocation)

- If the marshaled parameters are local objects, they are passed by *copy* using *object serialization* (writing the state of an object to a byte stream)
- If the parameters are remote objects, they are passed by *reference*

Pipes

- A pipe acts as a conduit allowing two processes to communicate
- Four issues in implementing a pipe
  - Does the pipe allow bidirectional communication, or is communication unidirectional?
  - If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?
  - Must a relationship (such as parent–child) exist between the communicating processes?
  - Can the pipes communicate over a network, or must the communicating processes reside on the same machine?

Some Pipe Types

- **Ordinary pipes**
  - Ordinary pipes are unidirectional, allowing only one-way communication
  - Ordinary pipes exist only while the processes are communicating with one another. On both UNIX and Windows systems, once the processes have finished communicating and have terminated, the ordinary pipe ceases to exist
- **Named pipes**
  - Communication can be bidirectional
  - No parent–child relationship is required
  - Once a named pipe is established, several processes can use it for communication
  - Additionally, named pipes continue to exist after communicating processes have finished