Chapter 3: Processes

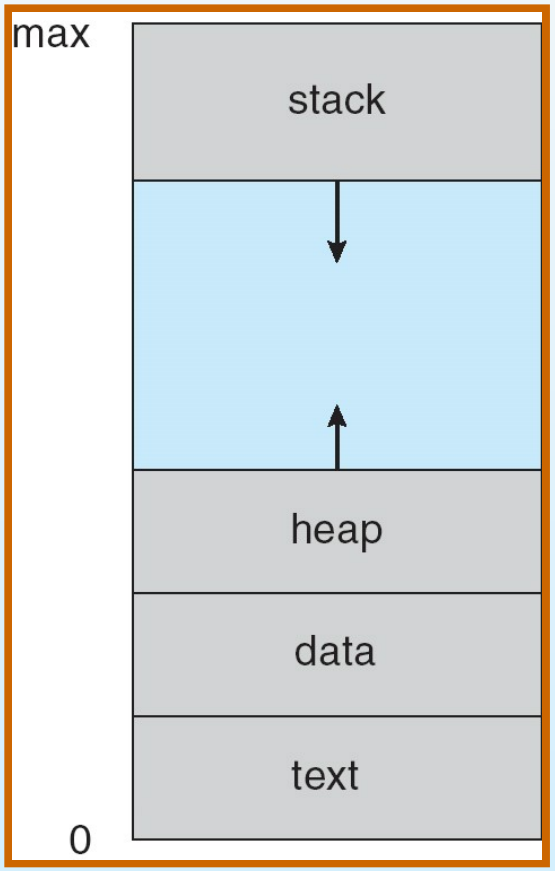
Chapter 3: Processes (进程)

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

Process Concept

- An OS executes a variety of programs:
  - Batch system – jobs (作业)
  - Time-shared systems – user programs or tasks
- Process
  - A program in execution
- A process includes:
  - program counter and other registers
  - text section
  - stack
  - data section
  - heap

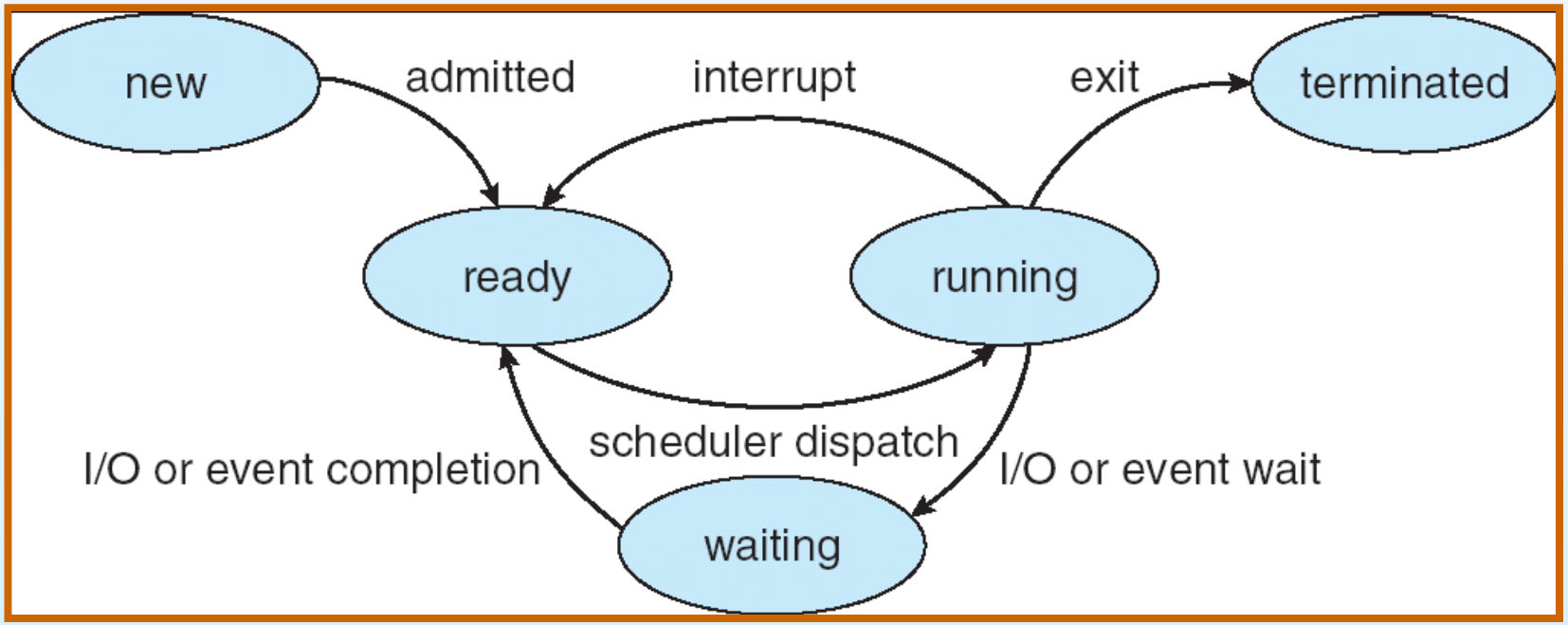
Process in Memory



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

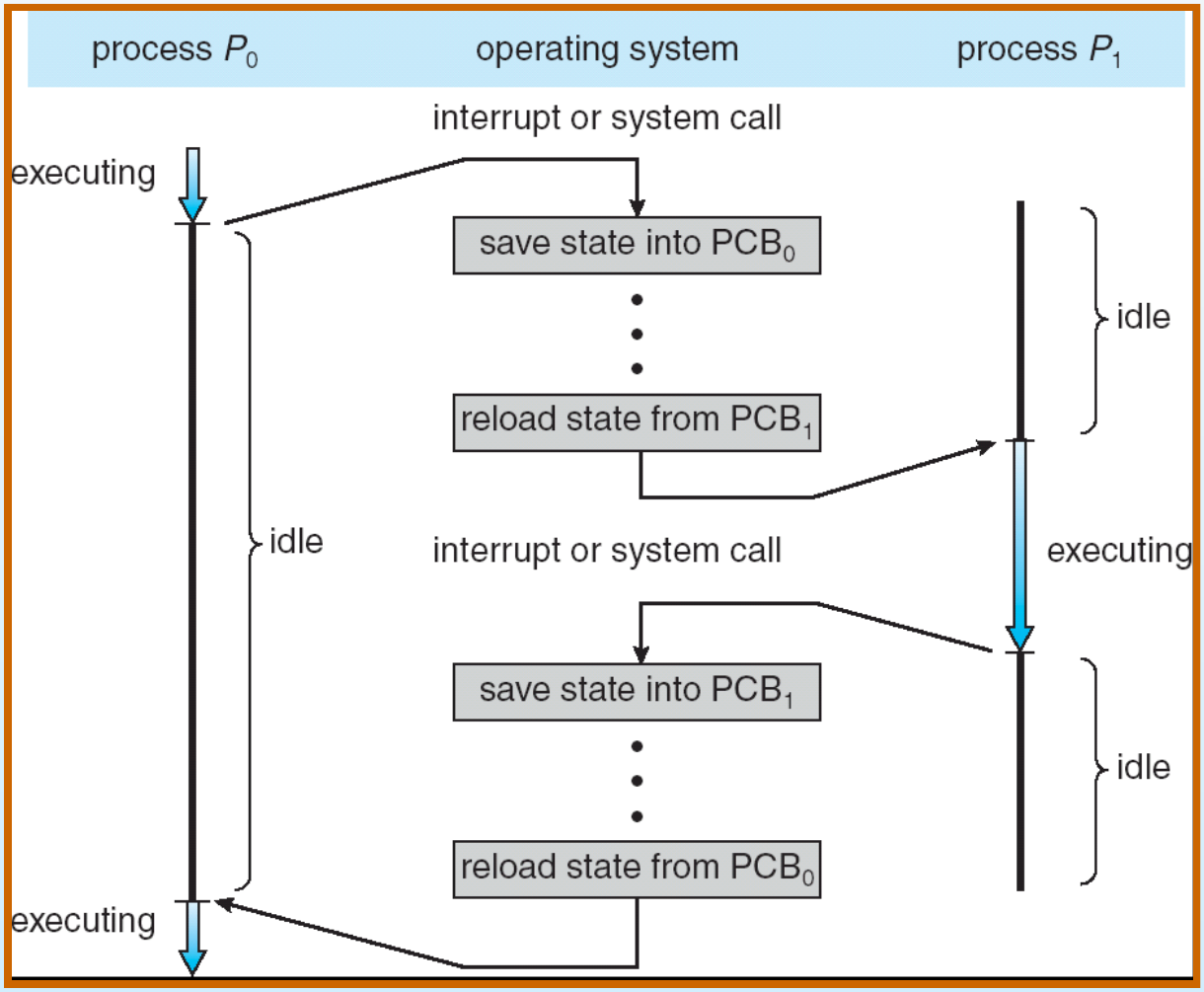


Process Control Block (PCB)

Information associated with each process

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

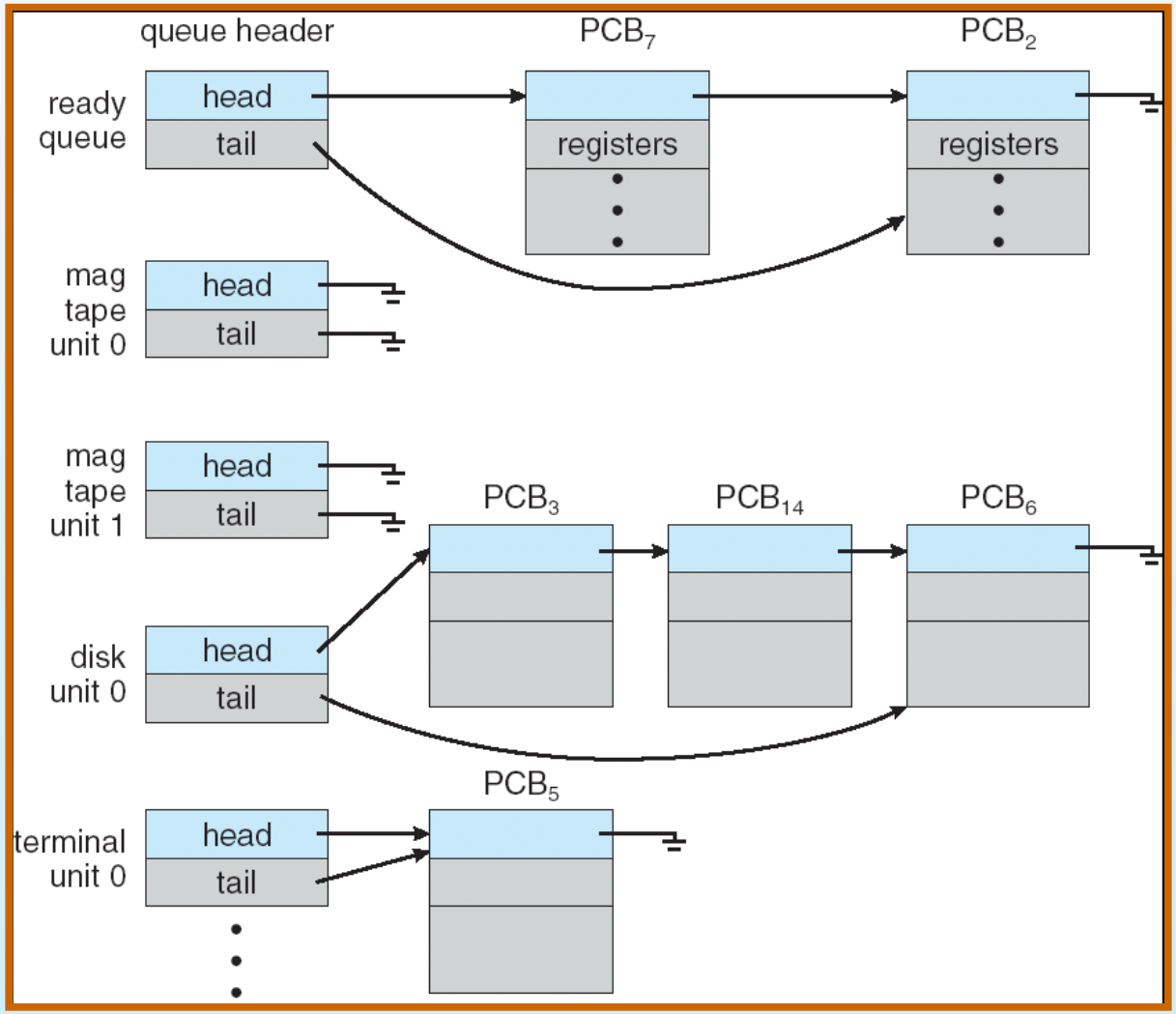
CPU Switch From Process to Process



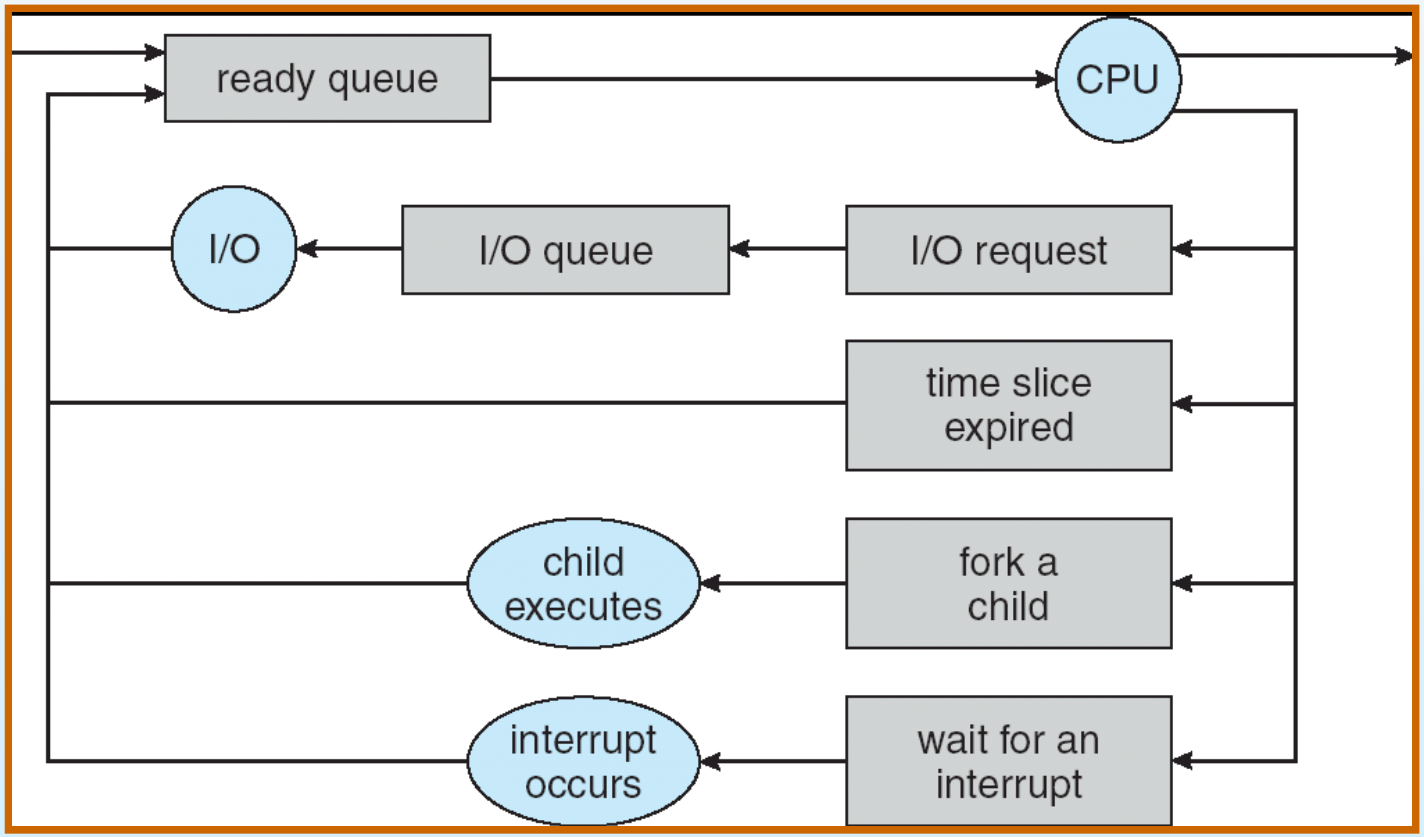
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



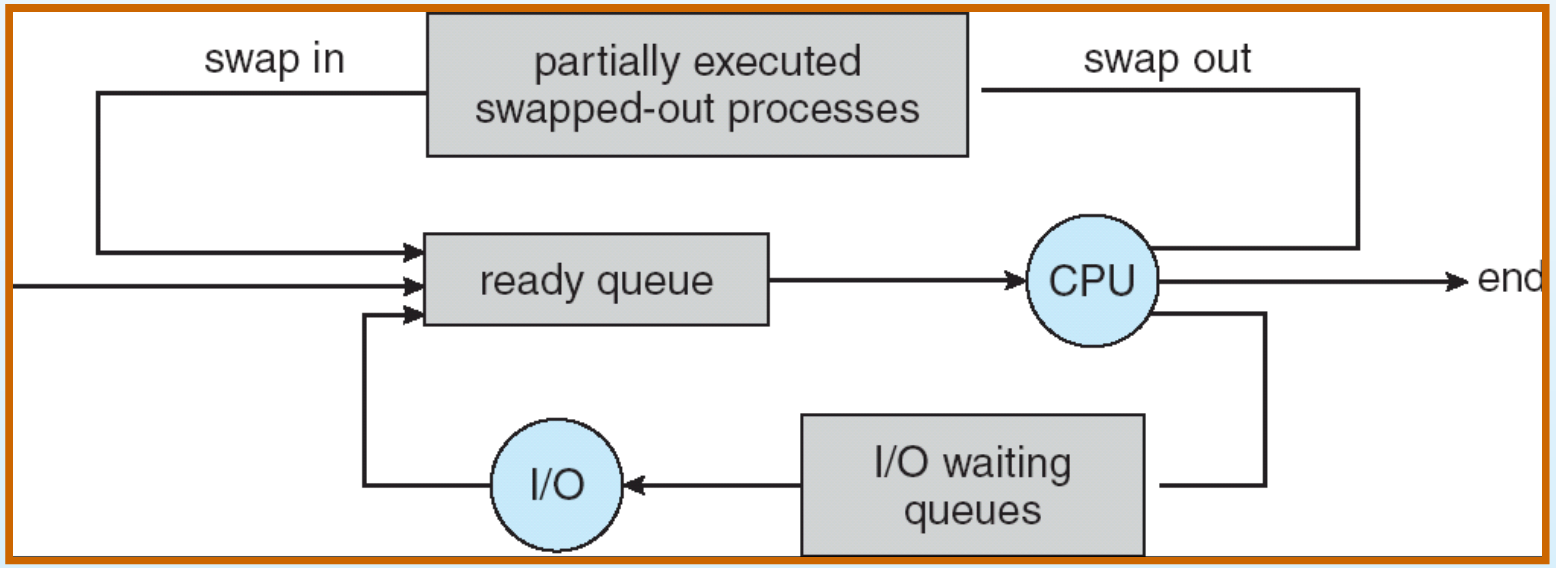
Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
- **Medium-term scheduler** (or swapping) – selects which process should be remove from memory

Addition of Medium Term Scheduling



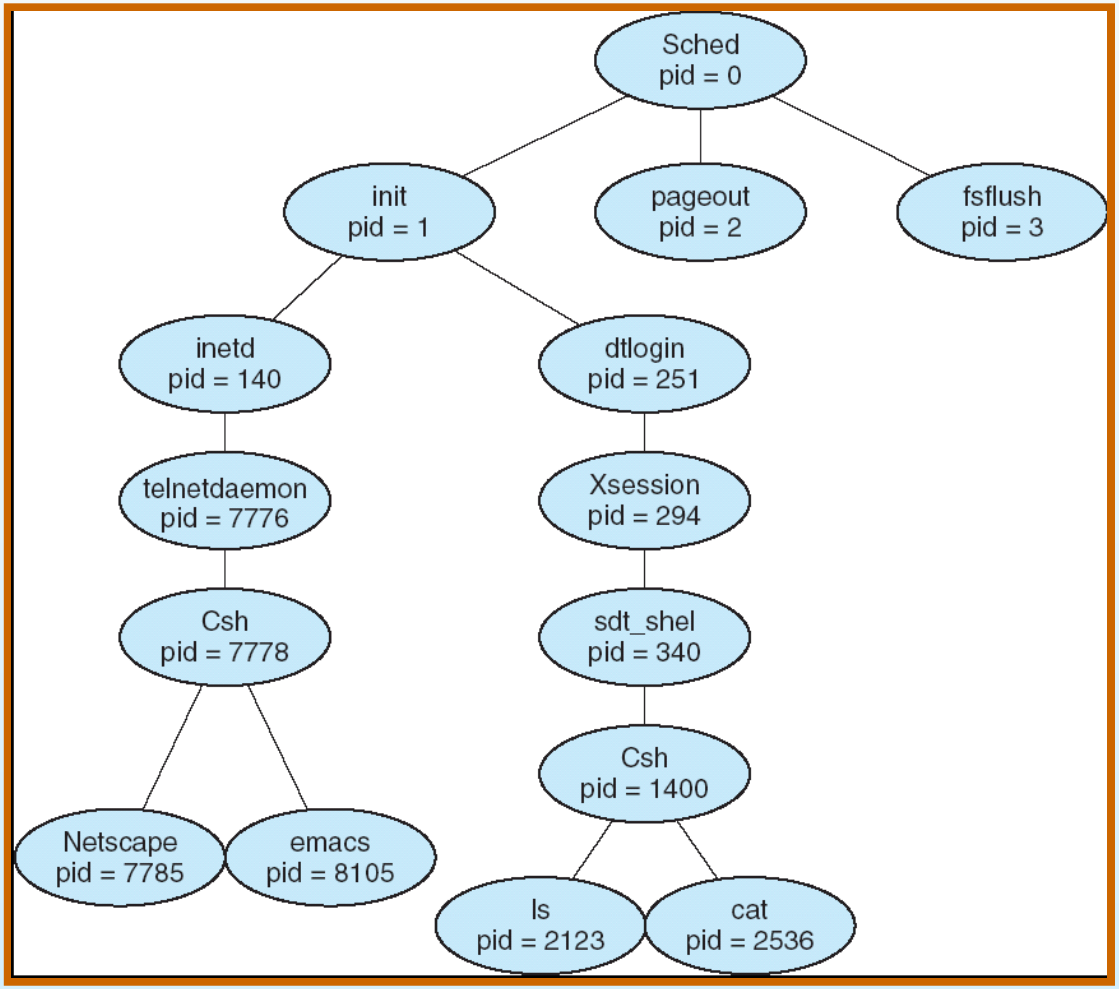
Context Switch

- When CPU switches to another process, the system must
  - Save the state of the old process
  - Load the saved state for the new process
- Context-switch time is overhead
  - The system does no useful work while switching
- Time dependent on hardware support

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

A tree of processes on a typical Solaris



Try “pstree -p” in Linux!

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

C Program Forking Separate Process

```
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 */
          execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
          /* parent will wait for the child to complete */
          wait (NULL);
          printf ("Child Complete");
          exit(0);
    }
}
```

Process Creation (Cont.)

- Windows examples
  - BOOL WINAPI CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
    );

C Program Creates Process in Win32

```
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 line
     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);
}
```

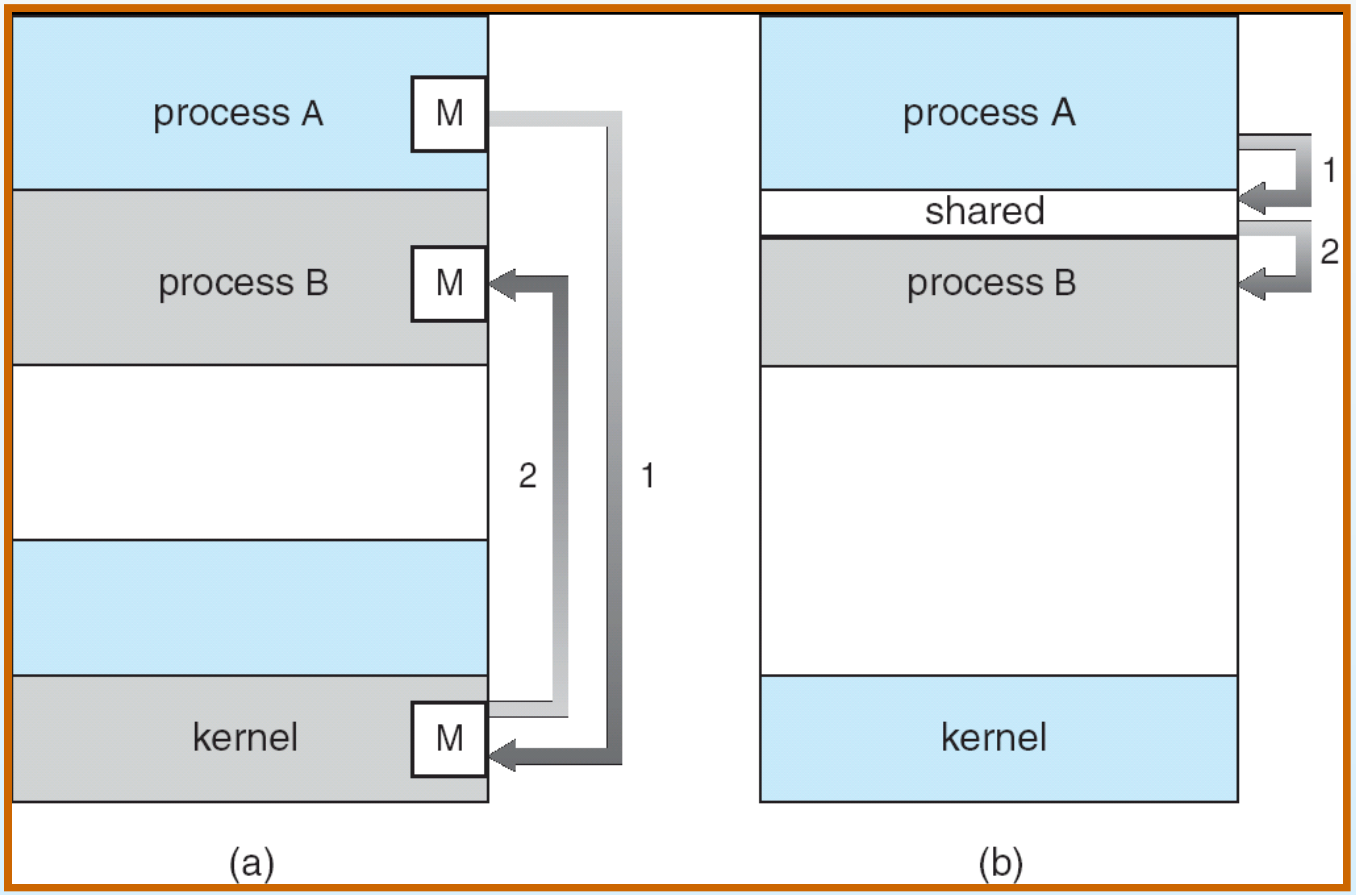
Process Termination

- Process executes last statement and asks the OS to delete it (exit)
  - Return value from child to parent (via wait)
  - Resources are deallocated by OS
- Parent may terminate execution of children (abort)
  - Kill any of the children as the parent likes
  - If parent is exiting
    - Some OS kill all children
    - Some OS give all children to *init*

Cooperating Processes

- **Independent** process can NOT affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Communications Models



(a) Message passing (b) Shared memory

Shared-Memory Systems

- OS prevents one process from accessing another process's memory
- Shared memory requires that processes agree to remove this restriction
- Processes' responsibility
  - Synchronization
  - Don't write to the same location simultaneously
- Not all OSes support this feature
- Some Oses are born shared-memory
  - Most of embedded systems

Shared-Memory in POSIX

- shmget()
  - Allocate a shared-memory segment
- shmat()
  - Attach the shared-memory segment
- shmdt()
  - Detach the shared-memory segment
- shmctl()
  - Control the shared-memory segment

Message-Passing Systems

- Processes communicate with each other without resorting to shared variables
- Particularly useful in a distributed environment
- Provides two operations:
  - send(*message*)
    - Message size fixed or variable
  - receive(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Synchronization vs. Asynchronous

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - The sender block until the message is received
  - The receiver block until a message is available
- Non-blocking is considered asynchronous
  - The sender send the message and continue
  - The receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   Sender must wait for receiver
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full
3. Unbounded capacity – infinite length
   Sender never waits

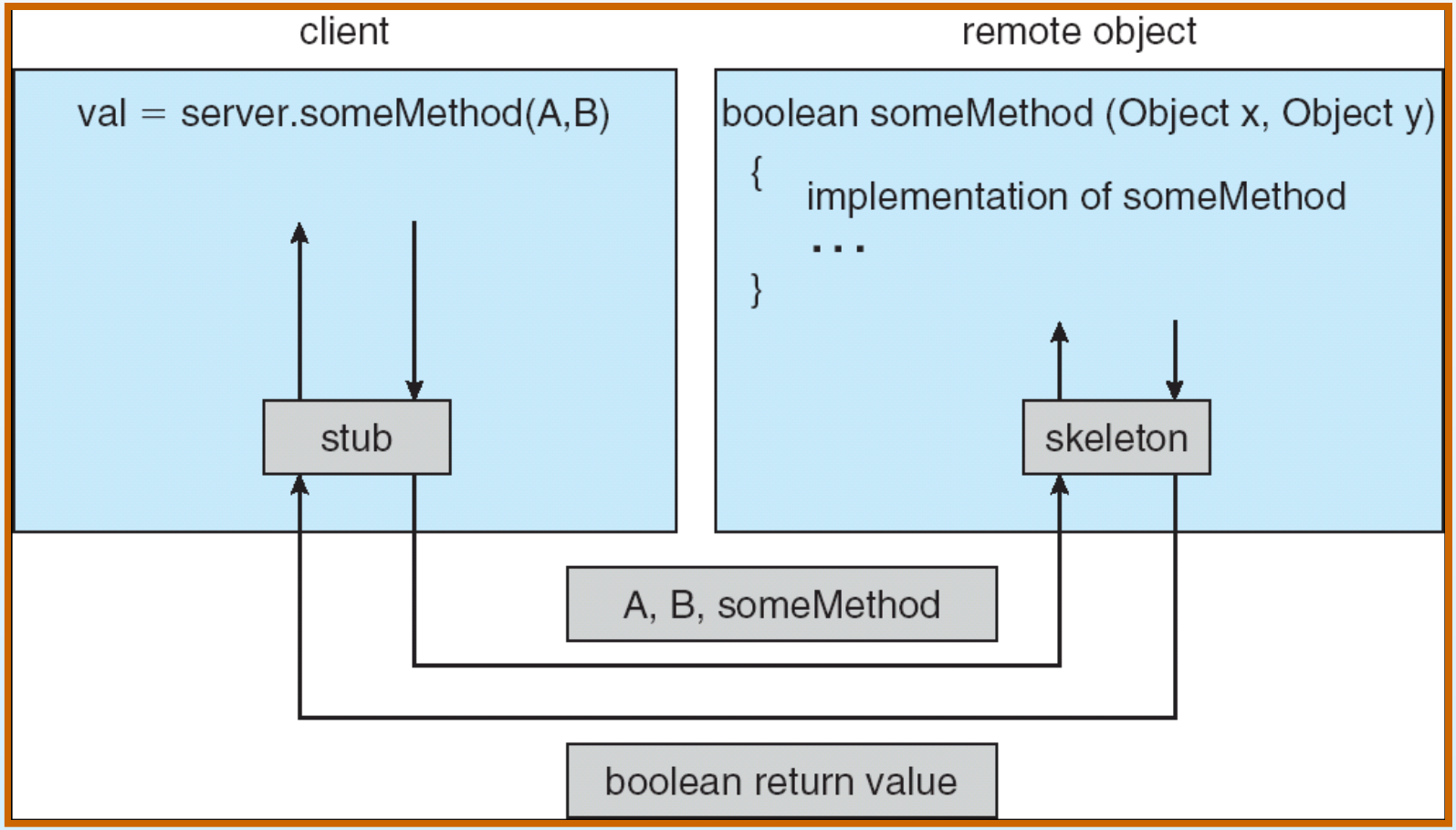
Client-Server Communication

- Sockets – leave to Network course
- Remote Procedure Calls
- Remote Method Invocation (Java)

Remote Procedure Calls

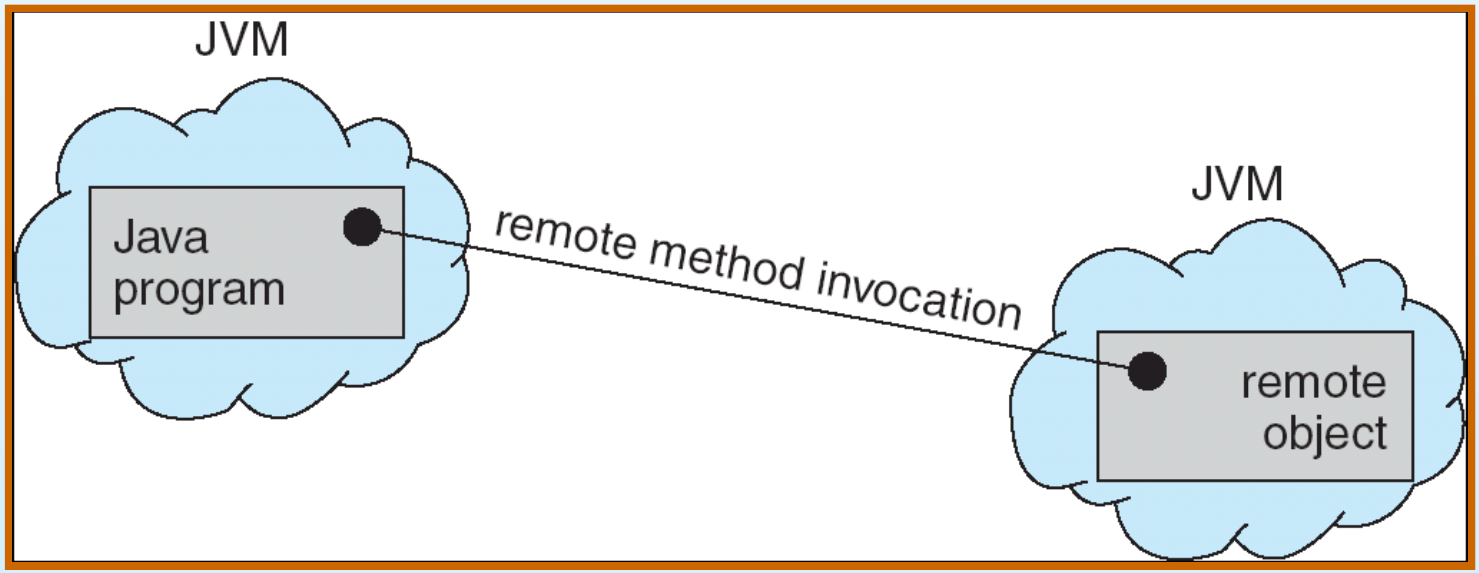
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- The client-side stub
  - Locates the server
  - *Marshalls* the parameters.
- The server-side stub
  - Receives this message
  - Unpacks the marshalled parameters
  - Performs the procedure on the server.

Marshalling Parameters



Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



End of Chapter 3