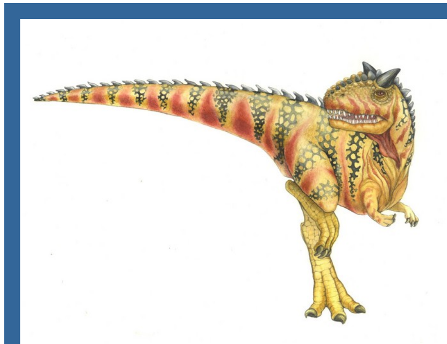
Theoretical Unit 2

(Book Chapters 3+4)

Processes and Threads

(edited & enhanced by rufino@ipb.pt, 2025/2026)



Unit 2: Processes & Threads

- Mono- vs Multi-programming
- Process Concept
- Operations on Processes
- Process Scheduling
- Inter-Process Communication
- Client-Server Communication
- Thread Concept
- Threading Models
- Threading Libraries
- Extra Topics

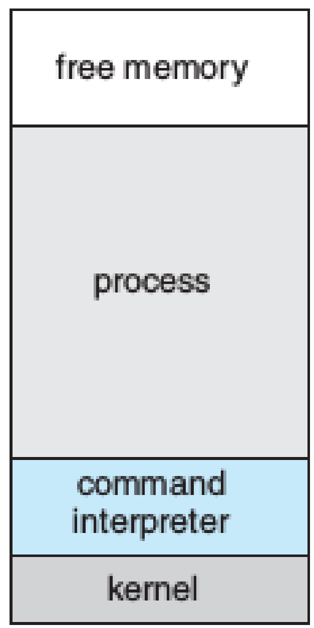
Theoretical Unit 2

2.1 Mono- vs Multi-Programming

Mono- vs Multi-Programming

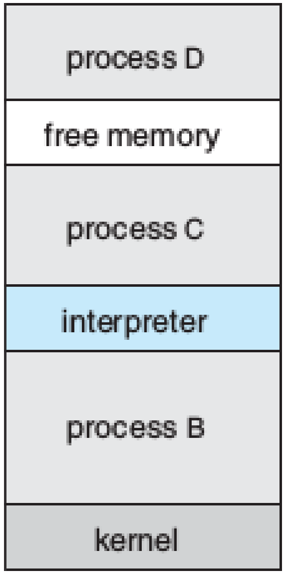
- Programs alternate between using CPU and IO devices
  - operating systems evolved in the way they manage this alternation
  - that evolution was also influenced by the underlying HW evolution

- **Mono-programming**: only one program in memory in addition to the OS
  - single program cannot keep both **single CPU** and I/O devices busy
  - a **single user** being attended; used in the first batch systems



Mono- vs Multi-Programming

- **Multi-programming**: several programs in memory in addition to the OS
  - when a program is blocked (e.g., for I/O), OS switches CPU to another
  - allows overlap of CPU and IO devices usage, maximizing utilization
  - the system may be **single/multi-user**, and **single/multi-CPU**
  - concurrent execution of many programs requires more evolved OS
- **Timesharing** (**multitasking**): fast task switching allows interactivity
  - an form of multi-programming supporting **interactive** computing



https://www.ibm.com/history/time-sharing

Theoretical Unit 2

2.2 Process Concept

Process Concept

- The **process** concept emerges in multi-programmed systems, from the need to ensure an efficient and safe co-existence of different programs

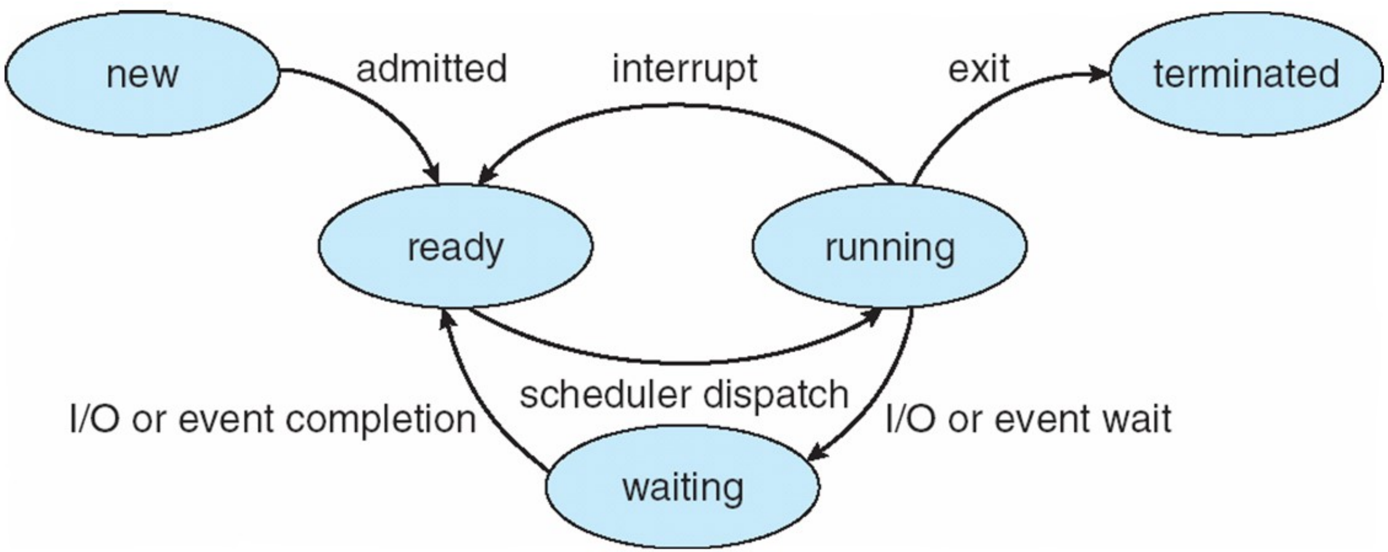
- **Program**: a set of **instructions,** and **data**, stored in an **executable file**
  - the type of instructions, and the internal format of an executable file, depends on the compiler, target architecture and hosting OS
  - a Program is a ***passive*** entity (like a recipe in a cookbook)
  - becomes a **Process** when executable file loaded into memory

- **Process**: “a program in execution”; includes following a certain path in the code, and all resources needed for the execution of that code
  - it is an ***active*** entity (though its code may not be always executing)

- The same program may exist in several processes; Processes may generate other clone processes (clones); Same process may execute different programs (one at a time); Processes may host special execution environments (e.g. JVM)

Process Life-Cycle

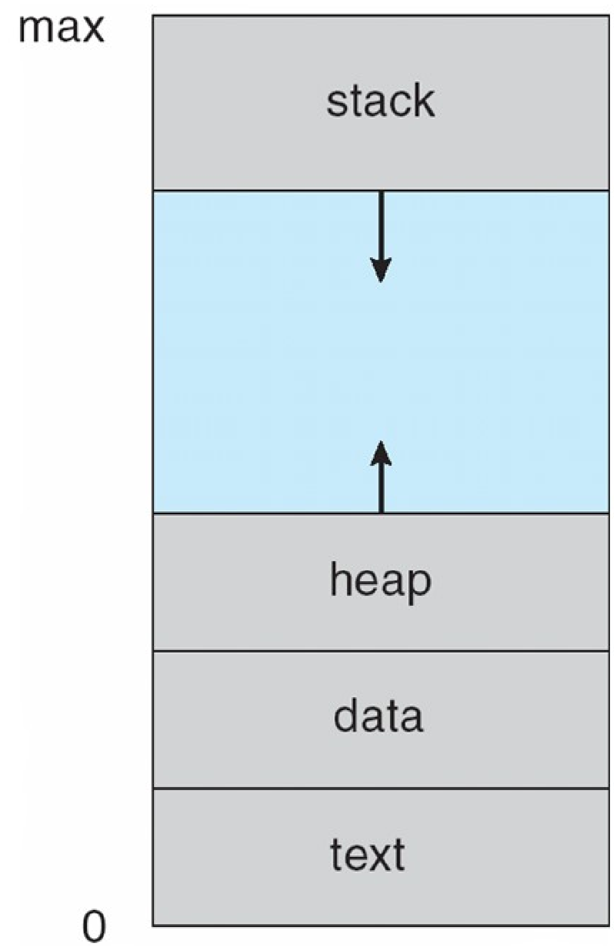
- As a process executes, it changes its **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



Process in Memory

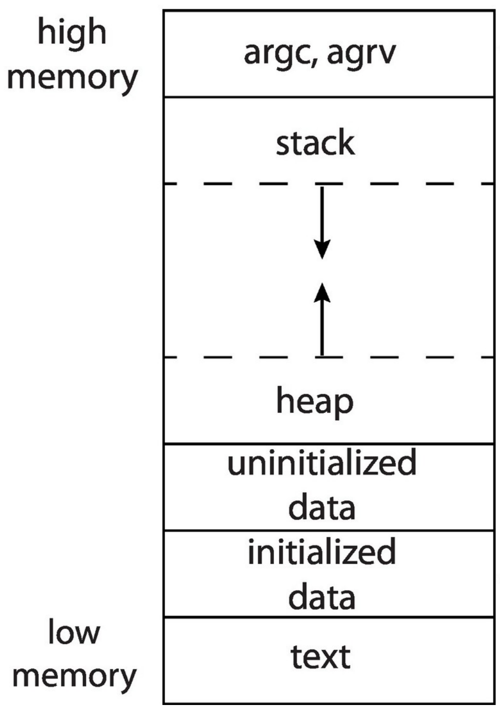
- At least two memory zones per process: i) a **user-level zone**; ii) a **kernel-level zone**

- The **user-level zone** includes segments for:
  - The program code or **text section**
  - A **Data section** for global variables and constants
  - The **Stack**, containing temporary data
    - Function parameters, return addresses, local variables (known as activation record / stack frame)
  - The **Heap**, containing memory dynamically allocated in run time



see also:
- https://www.youtube.com/watch?v=7dLZRMDcY6c
- https://www.youtube.com/watch?v=XbZQ-EonR_I

Memory Layout of a C Program




```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

The GNU `size` command can be used to determine the size (in bytes) of some of these sections. For the example program above it would yield:

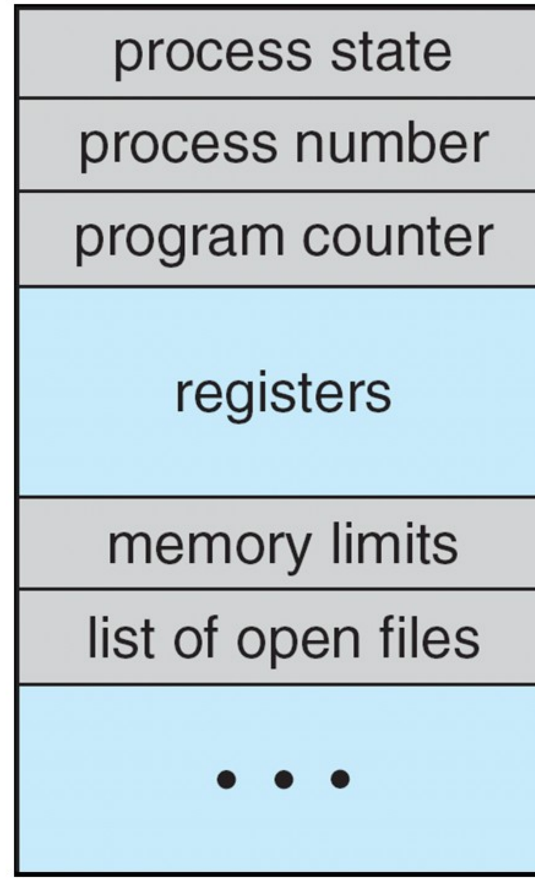
text	**data**	**bss**	**dec**	**hex**	**filename**
1158	**284**	**8**	**1450**	**5aa**	**example.exe**

text = code; `data` = uninitialized data; `bss` = block started by symbol (initialized data)

Process Control Block (PCB)

The **kernel-level zone** (task control block) stores specific attributes of each process :

- Process state – running, waiting, etc.
- Program counter – location of instruction to execute next
- CPU registers – contents (backup) of registers changed by the process
  - taken when switched out of the CPU, reloaded when re-entering the CPU
- 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

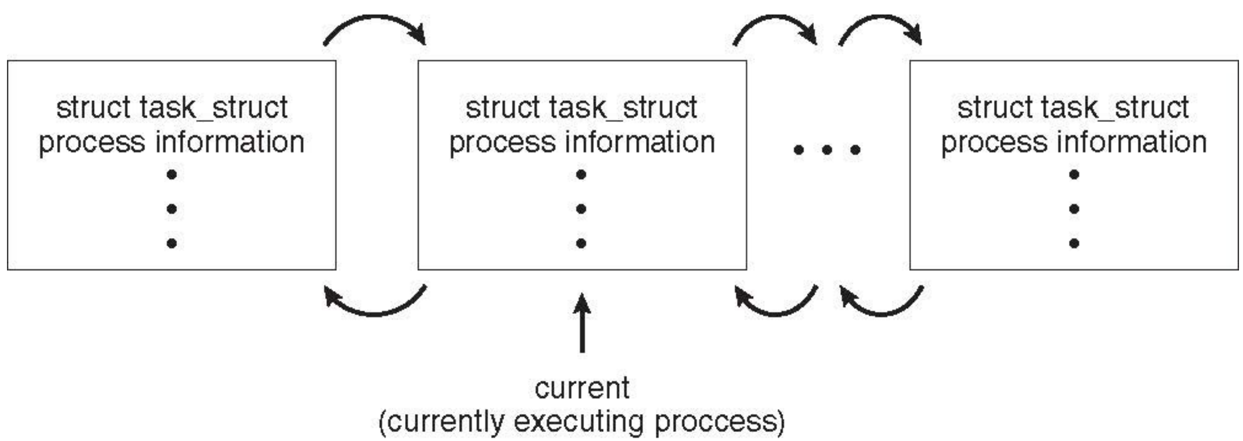


Process Representation in Linux

Represented by the C structure task_struct defined in <linux/sched.h>

```
char comm[TASK_COMM_LEN];    /* task command (executable name, excluding path) */
pid_t pid;                   /* process identifier */
long state;                  /* process state */
unsigned int time_slice      /* scheduling information */
struct task_struct *parent;  /* process parent */
struct list_head children;   /* process children */
struct files_struct *files;  /* open files information */
struct mm_struct *mm;        /* process address space */
void *stack;                 /* process stack */
int exit_code;               /* process exit code */
```

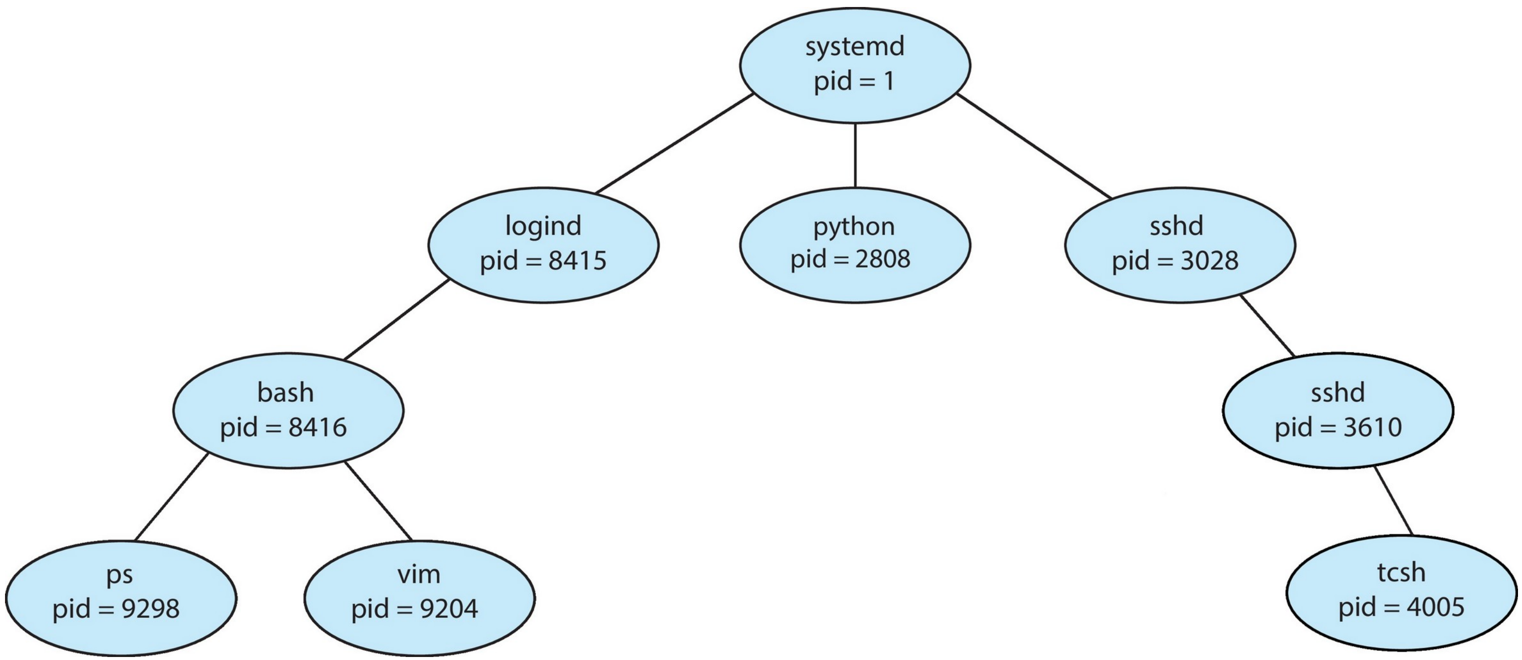
Process queues represented by double-linked lists of task_struct structures:



2.3 Operations on Processes

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Process identified and managed via a **process identifier** (**pid**)



A tree of processes in Linux (root process (pid=1) is `init` or `systemd`)

- Commands to list current processes in Linux: **ps**, **pstree**

Process Creation

- Resource sharing options
  - Parent and children share no resources
    - risk of overloading the system
  - Children share subset of parent's resources
    - risk of race conditions on shared data access

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

- Address space
  - Child is a duplicate of parent's
  - Child loads a new program into it

Process Creation (Cont.)

- UNIX example
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program

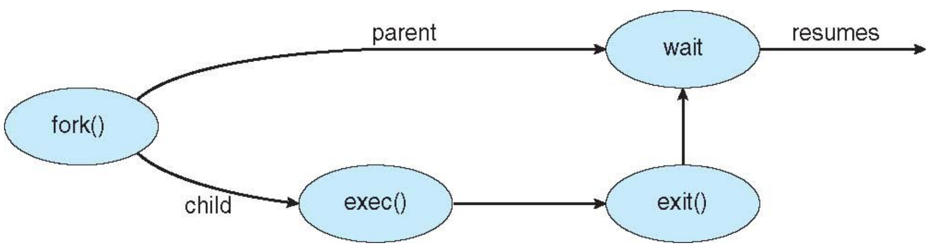
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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

- Windows example

```
#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

- **Voluntary termination**: process executes last statement and then asks the operating system to delete it using **`exit()`**
  - Returns status data from child to parent (via **`wait()`**)
  - Process resources are deallocated by operating system

- **Forced termination**: process is unexpectedly terminated by another process (via **`kill()`**) with enough privileges to do so

- Parent may terminate the execution of children processes once
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems (like VMS) do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **Cascading termination:**

    all children, grandchildren, etc. are terminated
  - The termination is initiated by the operating system

- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**), process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**

2.4 Process Scheduling

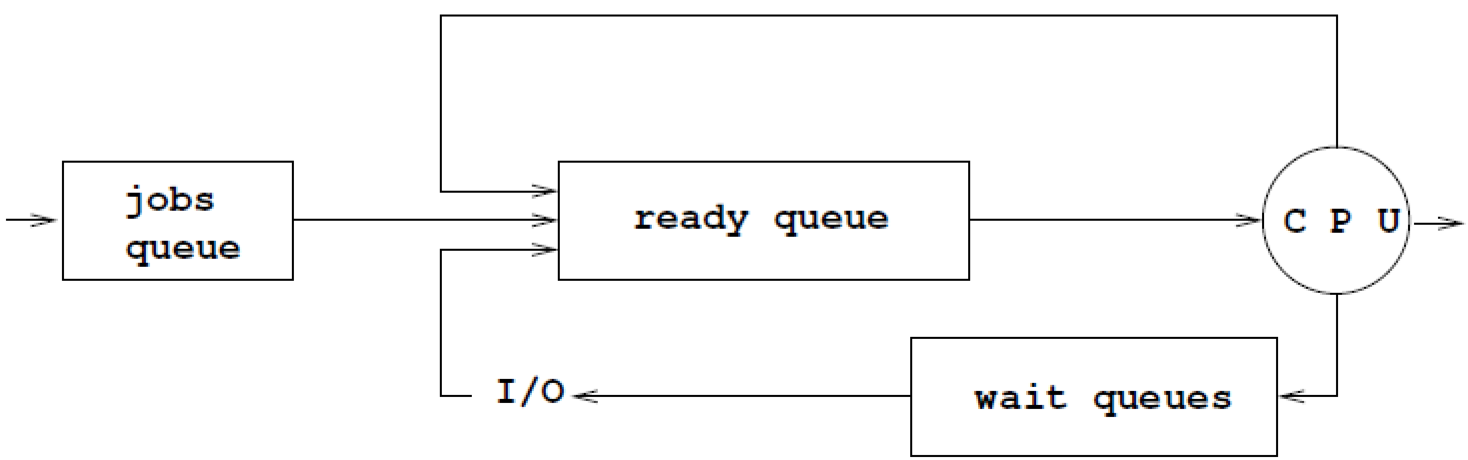
Process Scheduling

- **multiprogramming goal** is to maximize CPU use, ensuring at least one process per CPU-core executing at all times
  - also desirable to maximize usage of peripherals
  - **in time-sharing systems:** fast switch of CPU among different processes, allowing interactive use of the system by different users and providing the illusion of an exclusive system

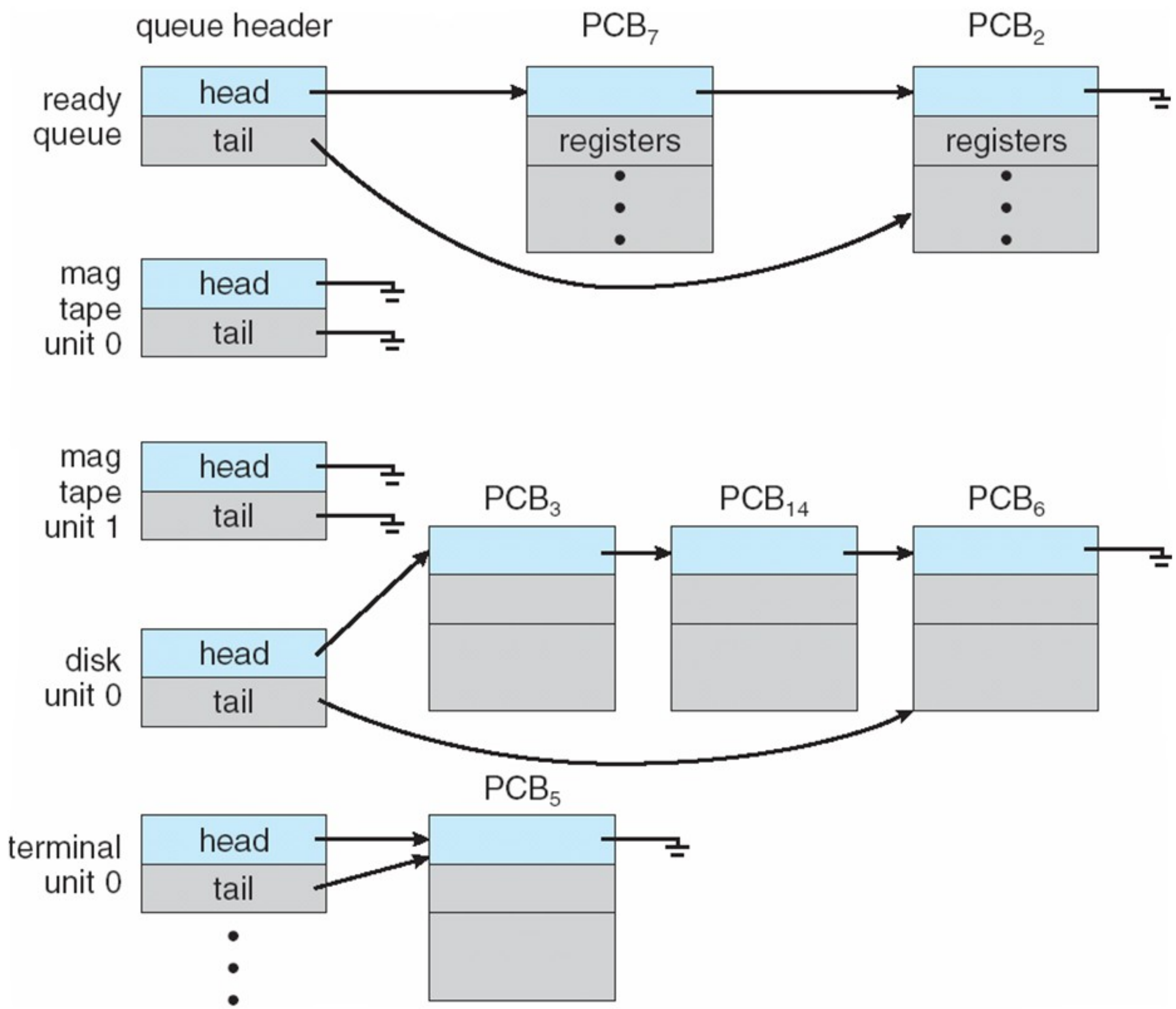
- to achieve these goals, the operating system implements:
  - **scheduling queues:** data structures used to organize processes currently in the same state of their life-cycle
  - **process schedulers**: OS code components that move processes between different queues (life-cycle states)

Process Scheduling

- **Process schedulers** maintains **scheduling queues** of processes
  - **Job queue** – set of all new processes in the system; <u>obsolete</u>
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting (blocked) for privileged services completion (system calls, I/O device requests, ...)
  - Processes migrate among the various queues

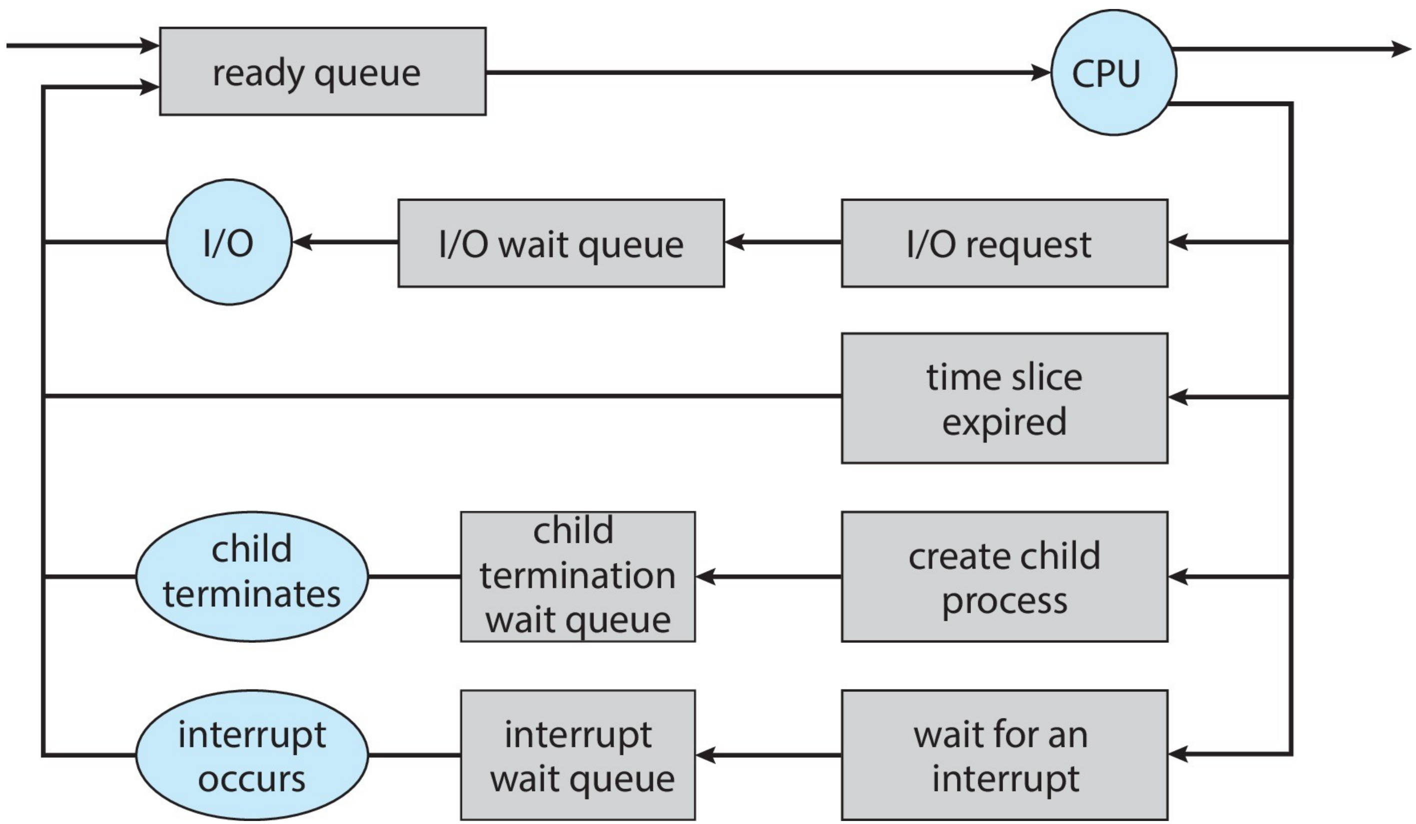


Ready Queue, I/O Device Wait Queues



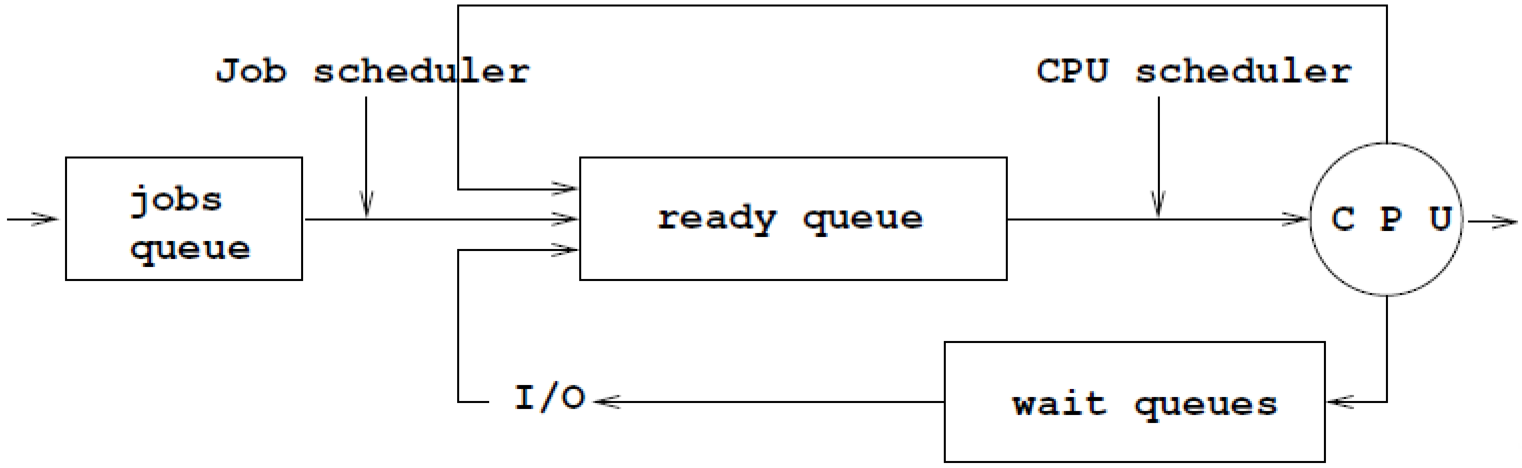
Representation of Process Scheduling

- **Queuing diagram** represents queues, resources, flows



Process Scheduling

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - It is invoked frequently (milliseconds) ⇒ must be fast
- **Long-term scheduler** (or **Job scheduler**) – selects which processes should be brought into the ready queue
  - It is invoked infrequently (seconds, minutes) ⇒ may be slow
  - The long-term scheduler controls the **degree of multiprogramming**



Process Scheduling

- Processes can be described as either:
  - **I/O-bound process**
    - spends more time doing (asking for) I/O than computations
    - many short CPU bursts
  - **CPU-bound process**
    - spends more time doing computations (and memory accesses)
    - few very long CPU bursts

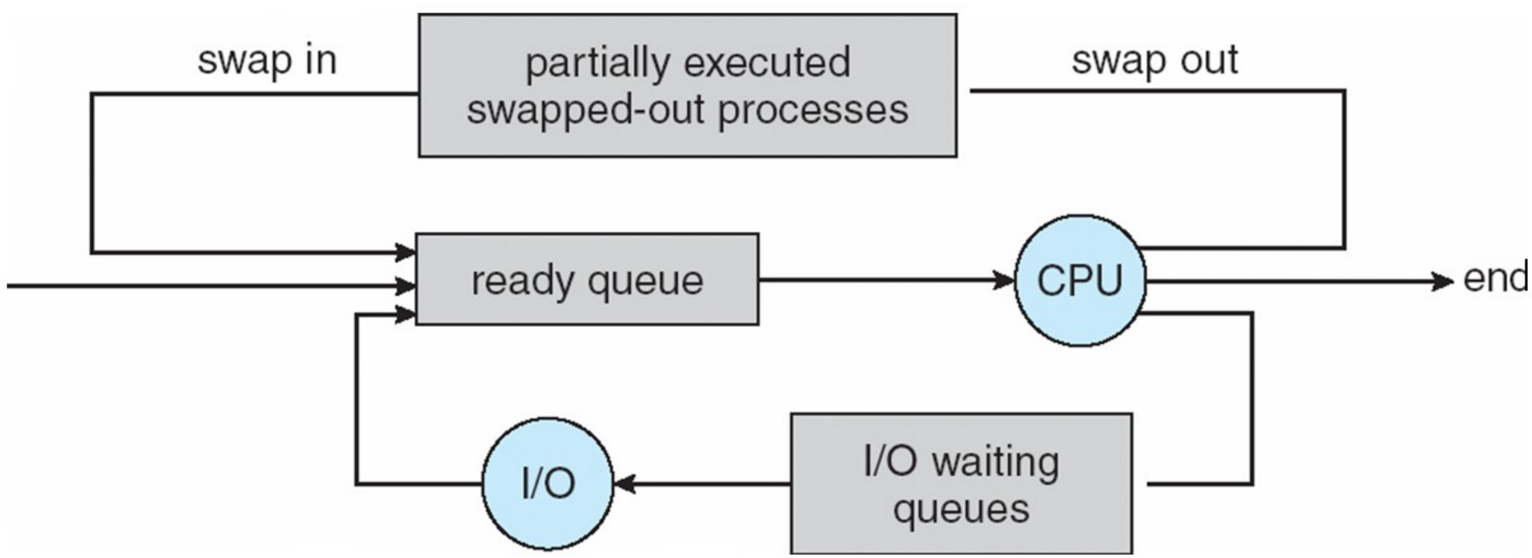
- Long-term scheduler strives for good ***process mix***
  - too many IO-bound processes

  => ready queue empty => low CPU utilization
  - too many CPU-bound processes

  => device queues empty => low device utilization

Medium Term Scheduling

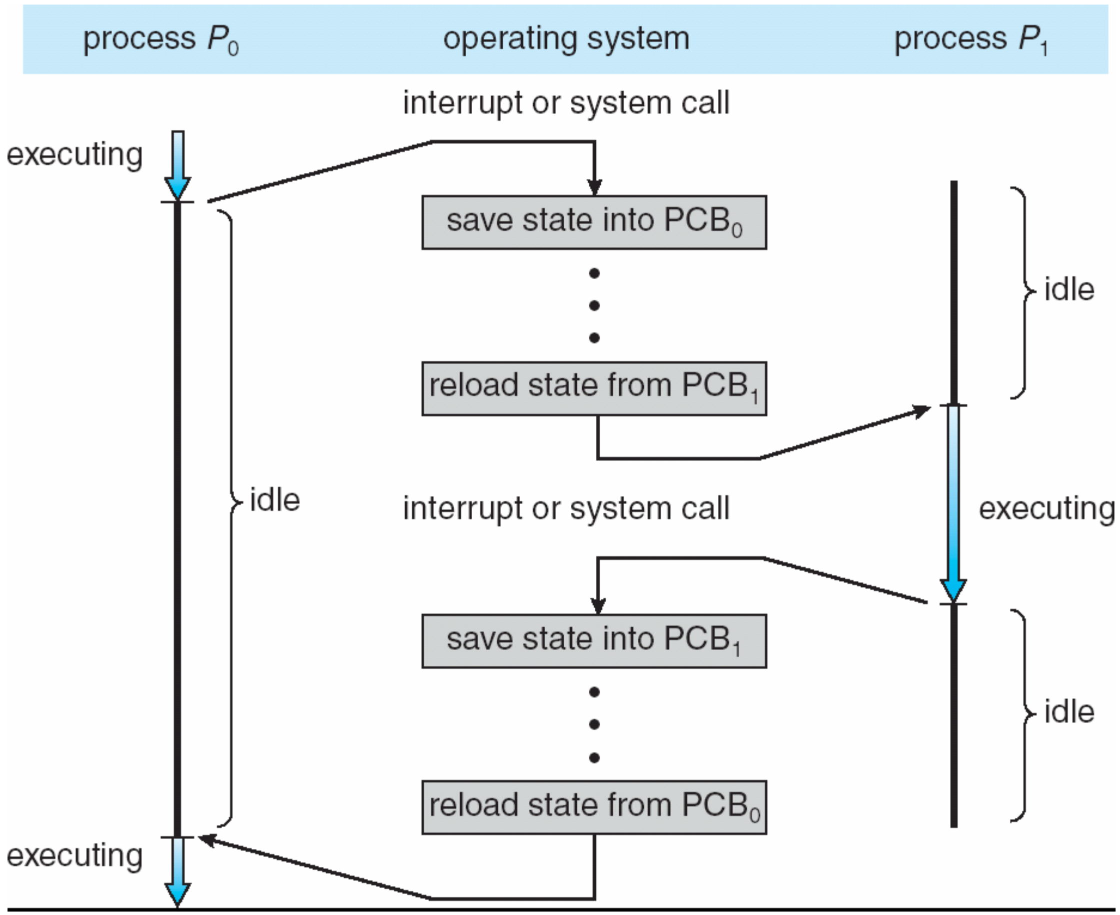
- **Medium-term scheduler** can be added if degree of multi-programming needs to decrease
  - This may be due to intense CPU or memory contention, or to improve the process mix
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process (in memory, or other registers) 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 overhead; 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
    - ➔ accelerate context switching by avoiding memory access

Context Switch



Theoretical Unit 2

2.5 Inter-Process Communication

Inter-Process Communication

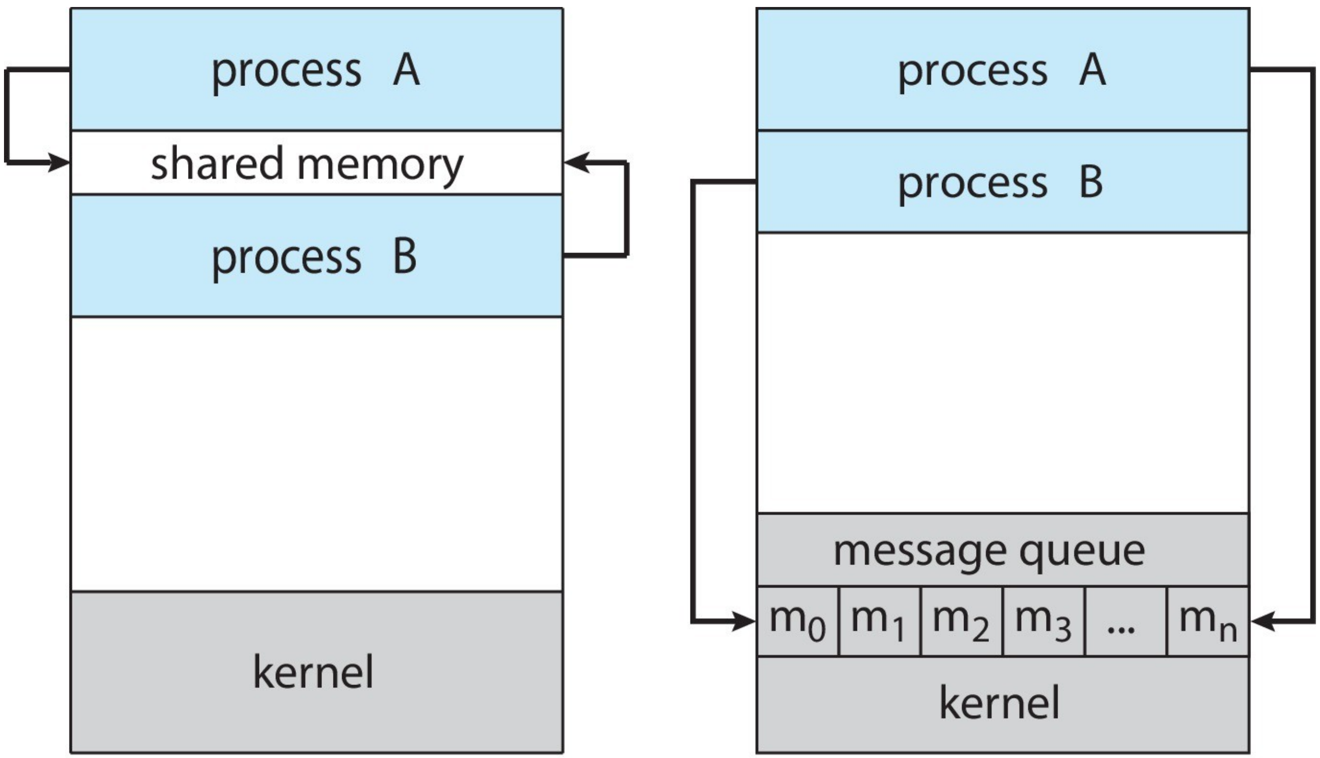
- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** processes cannot affect / be affected by others
- ***Cooperating*** processes can affect / be affected by others
  - in addition to changing behavior, this may also include sharing data (which, of course, may itself lead to a change in behavior)

- Reasons for cooperating processes:
  - **Information sharing** (e.g., copy+pasye, concurrent DB access)
  - **Performance increase** (parallel execution with several CPUs)
  - **Modularity** (split a task by several processes)

- Cooperating processes need **inter-process communication** (**IPC**)
  - to exchange data and/or synchronize

Communications Models

- Two main models of IPC: a) **Shared Memory;** b) **Message Passing**



(a) (b)

Communication Models

- Both models usually available, but fit different use cases

- **Message Passing:**
  - Communication through *syscalls* like send(...) and receive(...)
  - More suitable to (easier to implement in)
    - distributed environments
    - small data exchanges (implies frequent invocation of involved *syscalls*)

- **Shared Memory:**
  - Faster than message passing (*): needs only *syscall* to create shared zone; afterwards, access to shared memory is made by simply using pointers
    - (*) caveat: in multicore systems, several cores accessing the same shared memory block will invalidate each others caches ...
  - Demands explicit programming of all synchronization (see next example)

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - **producer** process produces information …
  - … that is consumed by a **consumer** process
  - this is done in a coordinated way (none can outrun the other)
- Assumes producer and consumer running at their own rhythm
- May be solved both using Message Passing or Shared Memory
- Two variants
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

```
// shared data:
#define N ...
typedef  ...  item_t; item_t buffer[N]; int in = out = 0;
```

```
// producer:
item_t nextProduced;
while (1) {
  produceItem(&nextProduced);
  while (((in + 1) % N) == out);
  buffer[in] = nextProduced;
  in = (in +1) % N;
}
```

```
// consumer:
item_t nextConsumed;
while (1) {
  while (in == out);
  nextConsumed = buffer[out];
  out = (out +1) % N;
  consumeItem(&nextConsumed);
}
```

- Solution above is correct, but can only use N-1 buffer items

Inter-process Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- Communication under the control of the users processes, not the OS
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
  - the previous example (bounded-buffer case) shows how hard it is
  - but there are more general synchronization mechanisms (see Chp 5)

Examples of IPC Systems – SyS V

- UNIX SysV Non-Private Shared Memory: **creator + writer** example

```
#define MAXSIZE 128
main() {
   int shmid; key_t key = 0x12345678;
   char *buffer;

   shmid = shmget(key, MAXSIZE*2, IPC_CREAT | 0666 );
   buffer = (char*)shmat(shmid, NULL, 0);

   strcpy(buffer, "first sentence");

   strcpy(buffer+MAXSIZE, "second sentence");
}
```

synchronization issues:
here, the creator is the sole writer; but right after creation, other processes may attach to and read/write the shared memory zone

Examples of IPC Systems – SyS V

- UNIX SysV Non-Private Shared Memory: **reader + destructor** example

```
#define MAXSIZE 128
main() {
  int shmid; key_t key = 0x12345678;
  char *buffer;

  shmid = shmget(key, MAXSIZE*2, 0666);
  buffer = (char*)shmat(shmid, NULL, 0);

  printf("%s\n", buffer);

  printf("%s\n", buffer+MAXSIZE);

  shmctl(shmid, IPC_RMID, NULL);
}
```

synchronization issues:
how does the reader know that it is safe to read from the shared memory zone (i.e., that the writer already wrote something there ?); even more, how is it sure that the shared memory even exists ?
strictly running the reader after the writer may be one solution

Inter-process Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Processes communicate with each other without resorting to shared variables
- Can be used both in shared-memory or distributed environments
  - Program once, run in both kind of systems (including hybrid)

- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

- The *message* size is either fixed or variable
  - Fixed-size messages make implementation easier, usage difficult
  - Variable-size messages make implementation difficult, usage easier

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 of communication link
  - Physical: Shared memory, Hardware bus, Network
  - Logical: Direct/indirect, Synchronous/asynchronous, Automatic/explicit buffering
- 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?

Message Passing: Naming: Direct Communication

- **Symmetric variant**: processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q

- Example: producer-consumer (with **blocking** communication – see slide 45)

```
// producer:
item nextProduced;
while (1) {
  produceItem(&nextProduced);
  send(consumer,&nextProduced);
}
```

```
// consumer:
item nextConsumed;
while (1) {
  receive(producer,&nextConsumed);
  consumeItem(&nextConsumed);
}
```

Message Passing: Naming: Direct Communication

- **Asymmetric variant**: only the sender names the receiver
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*id, message*) – receive a message from any process (*id* will store the identification of the sender process)

- Properties of communication link
  - Links are established automatically (only IDs are needed)
  - 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

Common disadvantage: low modularity (changing P or Q implies recompiling)

Message Passing: Naming: Indirect Communication

- Messages are exchanged through *mailboxes* (also named as *ports*)
  - **`send`** (A, message) – send a message to mailbox A
  - **`receive`** (A, message) – receive a message from mailbox A
  - 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
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

Message Passing: Naming: Indirect Communication

Mailbox sharing

- Problem:
  - 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.

Message Passing: Synchronization

- Message passing may be either **blocking** or **non-blocking**
- **Blocking** is considered **synchronous**
  - **Blocking send** -- sender blocks until the message is delivered (to the final receiver process, or to a mailbox/port)
  - **Blocking receive** -- receiver blocks until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- sender sends the message and continues, without awaiting for delivery confirmation (it may not be delivered)
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Message Passing: Buffering

- A link supports a maximum number of messages
- That number is the capacity of the link attached message queue
- message queue 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
- For non-zero capacity queues, confirming the reception of a message involves additional messages; for instance:
  1. sender P executes

     "send(Q, message); receive(Q, message);"
  2. receiver Q executes

     "receive(P, message); send(P, ACKNOWLEDGE);"

Examples of IPC Systems - Pipes

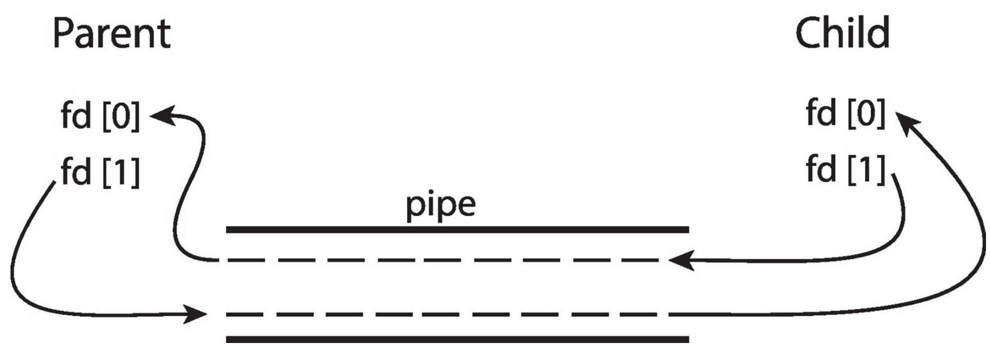
- Acts as a conduit allowing two processes to communicate

- 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?

- Using pipes in simplex mode helps avoiding deadlocks …
- Pipes have limited buffering capacity (writing may block)

Examples of IPC Systems - 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
  - Require parent-child relationship between communicating processes
    - typically, a parent process creates a pipe and uses it to communicate with a child process that creates right-after



- Windows calls these **anonymous pipes**

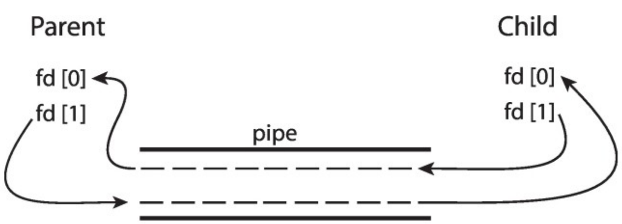
Examples of IPC Systems - Ordinary Pipes

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }
```




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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the read end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

synchronization issues:
if pipe is empty, the reader will block (reception is synchronous)

Examples of IPC Systems - Named Pipes

- **Named Pipes** are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary between the communicating processes
  - Several processes can use the named pipe for communication
  - Provided on both UNIX and Windows systems

Theoretical Unit 2

2.6 Client-Server Communication

Client-Server Communication

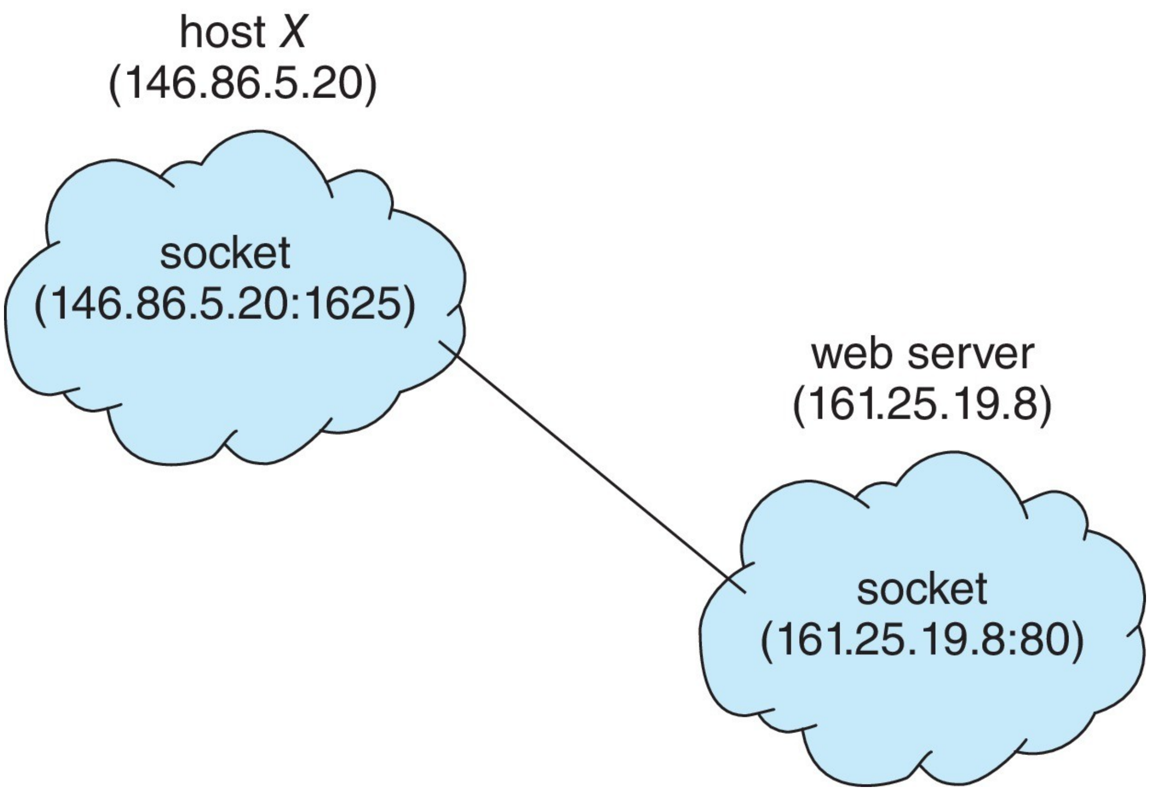
- IPC may be used only for processes within the same machine
- processes running in different machines must use other mechanisms (these may also be used within the same machine, but less efficiently)

- Client-Server communication mechanisms include, among others
  - Sockets
  - Remote Procedure Calls
  - Remote Method Invocation (Java)

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
- All ports below 1024 are ***well known***, used for standard services
- support both *connectionless* and *connection-oriented* communication
- provide efficient communication, but require low-level programming
- data is seen as an unstructured byte sequence; it is up to the programmer to define sizes and bounds to give it structure !

Socket Communication



Socket Communication

- Sockets BSD: TCP Client-Server Example

```c
#include "inet.h"
int main() // a TCP server
{
 int sockfd, newsockfd; socklen_t clilen;
 struct sockaddr_in cli_addr, serv_addr;
 char buffer[MAXLINE];

 // create a TCP socket
 sockfd=socket(AF_INET, SOCK_STREAM, 0);

 // bind the socket to a local port
 bzero(&serv_addr, sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
 serv_addr.sin_port = htons(SERV_TCP_PORT);
 bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

 // set max pending connections (5)
 listen(sockfd, 5);

 while(1) {
   // wait for connection
   clilen=sizeof(cli_addr);
   newsockfd=accept(sockfd, (struct sockaddr*)&cli_addr, &clilen);

   // fork a child to deal with the request
   switch(fork()) {
     case 0: // CHILD
             close(sockfd);
             printf("Client Address: %s\n", inet_ntoa(cli_addr.sin_addr));
             printf("Client Port: %d\n", ntohs(cli_addr.sin_port));
             read(newsockfd, buffer, MAXLINE);
             printf("Client Data: %s\n", buffer);
             close(newsockfd);
             exit(0);
     default: // PARENT
              close(newsockfd);
   }
 }
}
```

```c
// inet.h
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#define SERV_TCP_PORT 7564
#define SERV_HOST_ADDR "127.0.0.1"
#define MAXLINE 512
```

```c
#include "inet.h"
int main(int argc, char **argv) // a TCP client
{
 int sockfd;
 struct sockaddr_in serv_addr;

 // create a TCP socket
 sockfd=socket(AF_INET, SOCK_STREAM, 0);

 // set server address and port
 bzero(&serv_addr, sizeof(serv_addr));
 serv_addr.sin_family = AF_INET;
 serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
 serv_addr.sin_port = htons(SERV_TCP_PORT);

 // connect to server
 connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

 // send message
 write(sockfd, argv[1], strlen(argv[1])+1);

 // close connection
 close(sockfd);

 return(0);
}
```

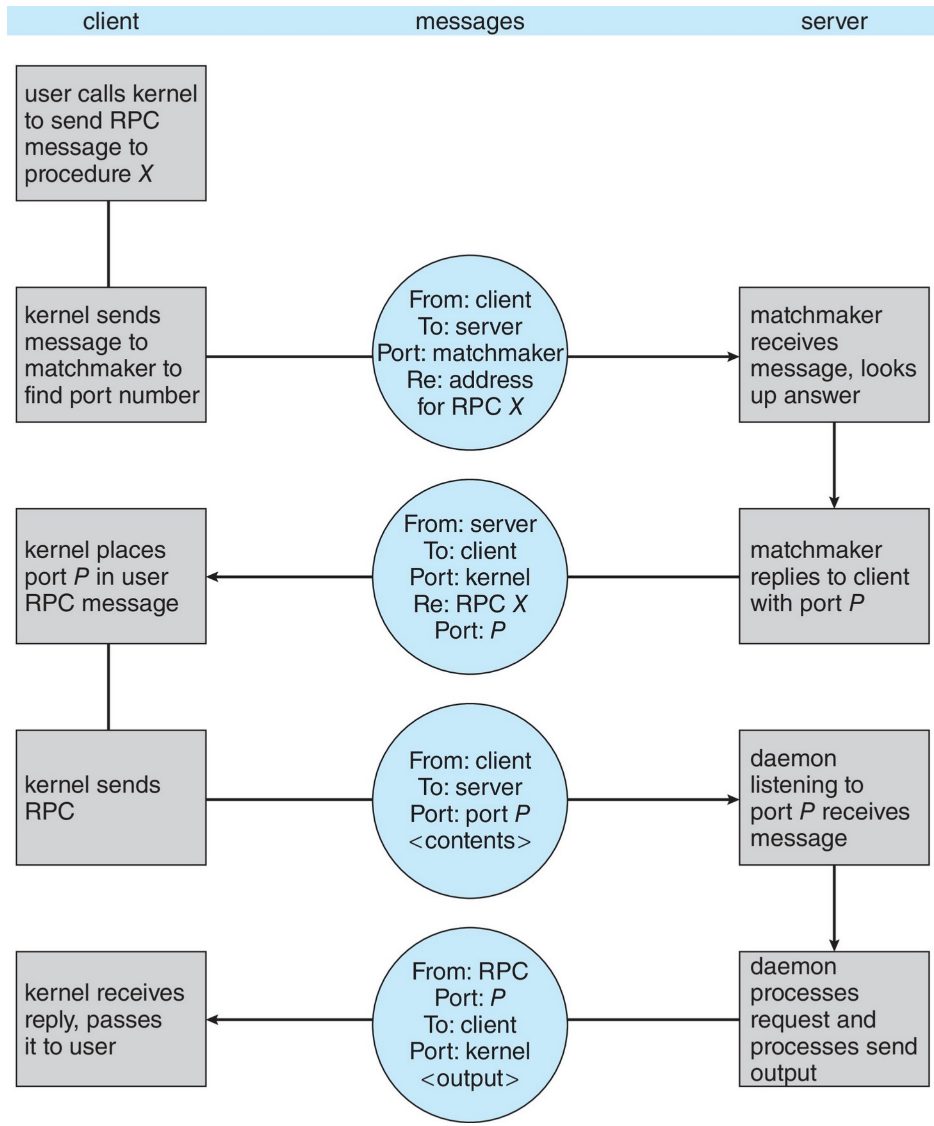
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – automatically generated proxy code (local and remote)
- **Client-side stub:**

  \- locates the server, **marshalls** the parameters, sends request
- **Server-side stub:**

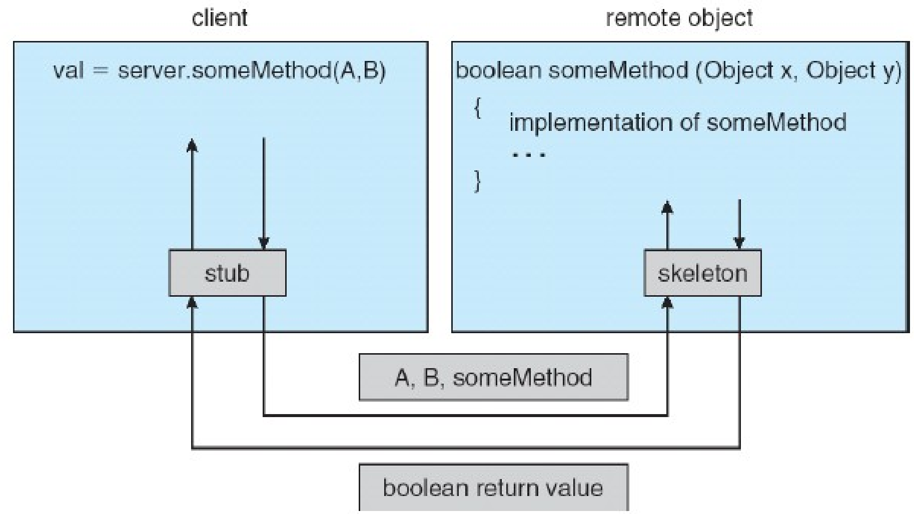
  \- receives request, unpacks the marshalled parameters, performs the procedure on the server, and builds and sends back the answer
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures - **big-endian x little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS provides a rendezvous (**matchmaker**) service to connect endpoints

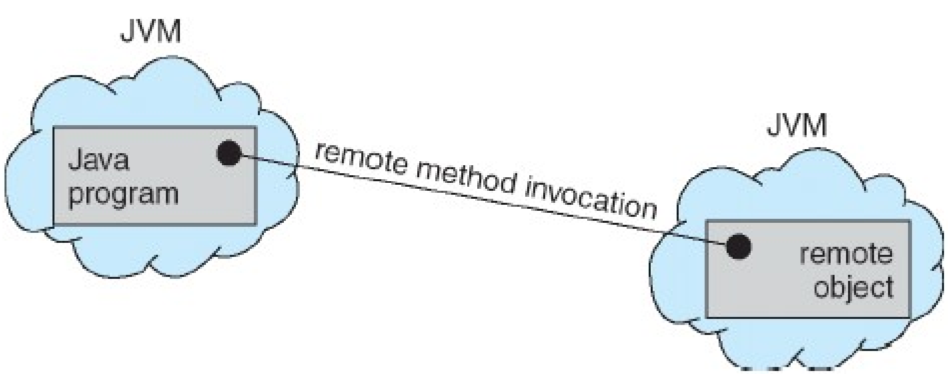
Execution of RPC



Remote Method Invocation (Java RMI)

- Similar to traditional (Sun) RPCs, but between Java Virtual Machines
- Sun RPCs best for procedural programming; Java for OO programming
- JAVA RMI allows
  - invoking remote object methods
  - passing objects as parameters

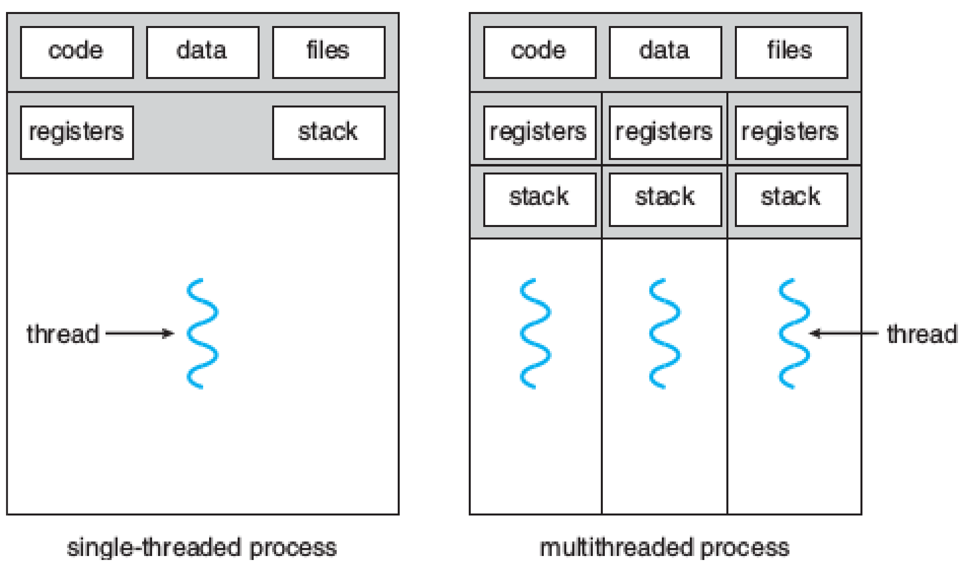




2.7 Thread Concept

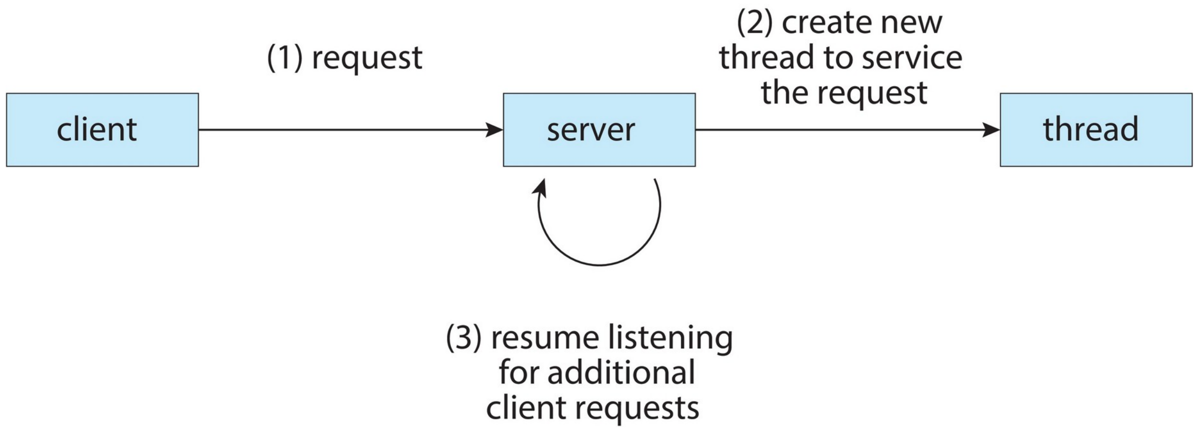
Thread Concept

- **Thread** (or Light-Weight Process): basic unit of CPU utilization
  - a path in the code and the resources needed to follow that path
  - resources specific to a thread: TID, CPU registers (PC, …), stack
- a **Process** (Heavy-Weight Process) includes at least one thread – the main thread (C programs: the thread executing the `main` function)
  - a process may have one or several threads (single/multi-threaded process)
  - threads of the same process share its PID, code, global data, heap, files, ...



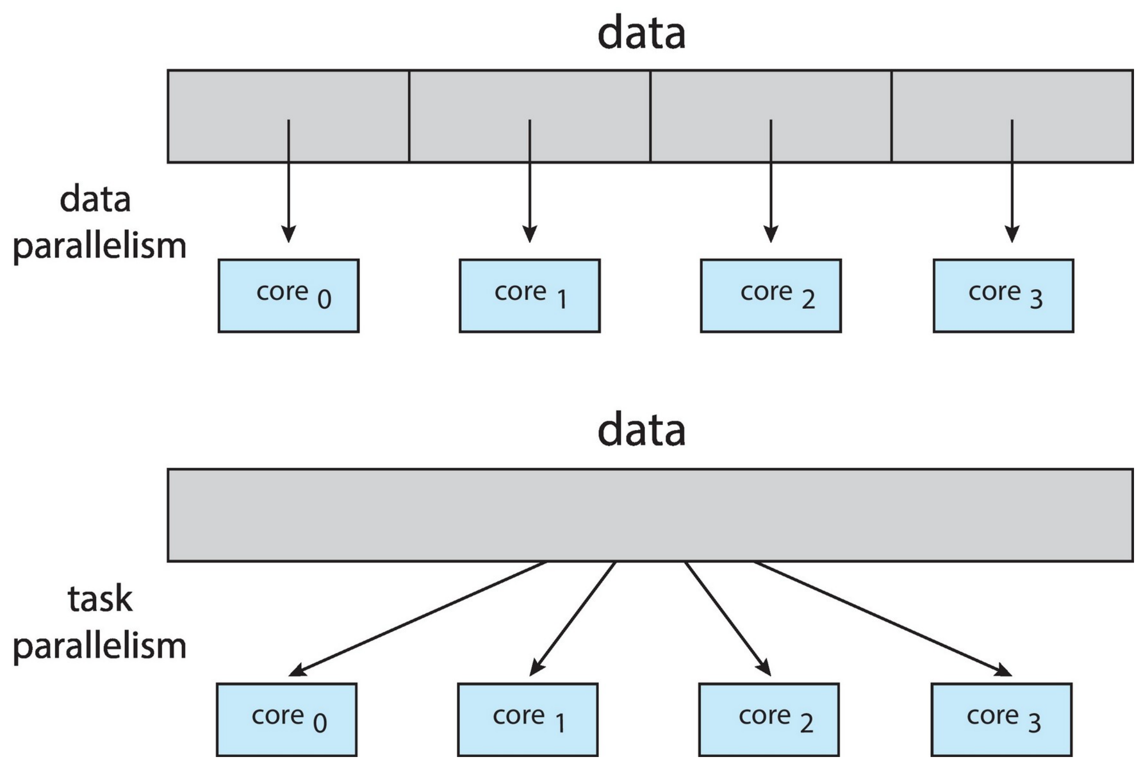
Motivation and Examples

- Nowadays, **most applications are multi-threaded** (and also the OS), which helps taking advantage of the modern multi-core CPUs
- An **image processing app** (create thumbnails, apply filter, etc.) may use several threads per image, per set of images (one thread per image) etc.
- A **web browser** may use on thread per tab, or several threads per tab (a thread to render the page, another to communicate with the web server)
- A **word processor** may use dedicated thread to render the document, react to the mouse and keyboard, spelling and grammar, IA assistant
- In a **web server**, a thread listens for new requests, and creates a new thread to service each request (or manages a pool of worker threads)



Motivation and Examples

- A computationally demanding problem may be solved faster by splitting it in (relatively) independent sub-problems and assign one or more threads to solve each sub-problem: the main idea behind **Parallel Computing**



Advantages and Disadvantages

- Some Advantages (vs Processes)
  - Need less resources (*n* threads are lighter than *n* processes)
  - Faster to create (need less state)
  - Faster to switch CPU among threads (less state to save/restore)
  - Easier data sharing (no need for shared memory syscalls)
  - Better responsiveness (one part may react while other part blocked)
  - Better scalability (more efficient use of multicore CPUs)

- Some Disadvantages (vs Processes)
  - multi-threading not necessarily faster than multi-process
  - access to shared data may lead to hard-to-debug bugs
  - not all programming languages (and tools) support threads (debugging threaded apps needs thread-aware debbugers)

Theoretical Unit 2

2.8 Threading Models

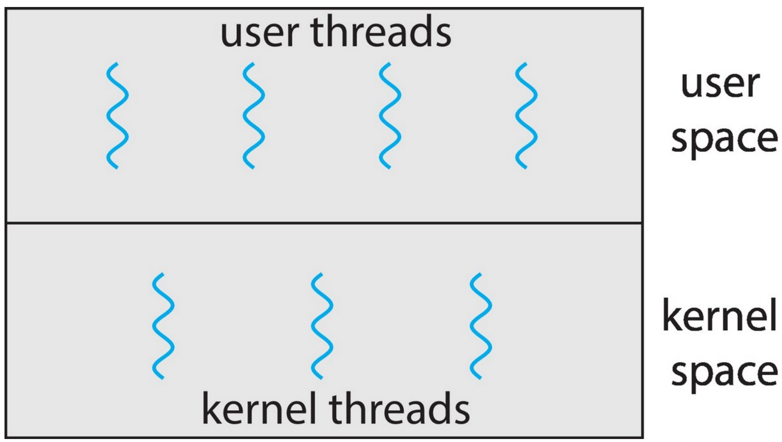
Threading Models

- **kernel thread**: we have been assuming a thread is a kernel-level feature (created and managed in the kernel), exposed to applications via syscalls

- **user thread**: but threads used by applications are usually abstractions created and managed by user-space libraries and exposed via APIs

- how to **map user (virtual) threads into kernel (real) threads** ?

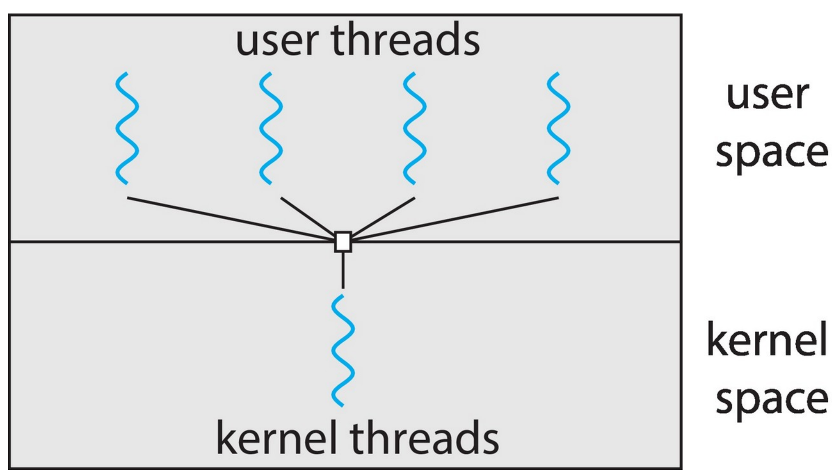
  - **many-to-one** model
  - **one-to-one** model
  - **many-to-many** model
  - **2-level** model



Threading Models

Many-to-One

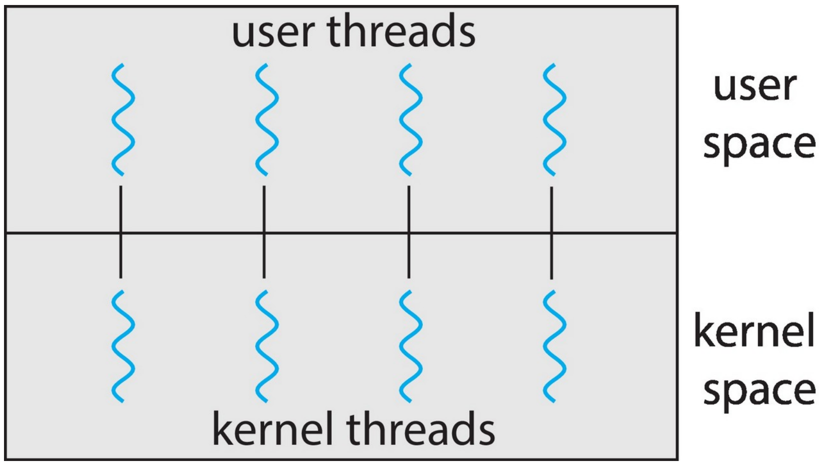
- maps many user threads into a single kernel thread
- allows efficient management of user threads by user-space library (alternating thread execution does not imply CPU context switching)
- the entire process blocks if one user thread calls a blocking syscall
- only concurrent execution (not parallel) of the many user threads
- example: Java Green Threads library used in the Solaris OS



Threading Models

One-to-One

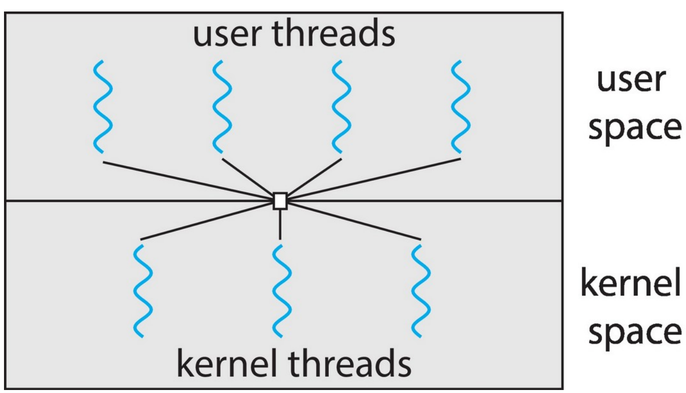
- maps each user thread into a single kernel thread
- vs many-to-one
  - supports more concurrency/parallelism
  - implies creating more kernel threads (too much may overload system)
- example: Windows and Linux, model exposed by the Pthreads library

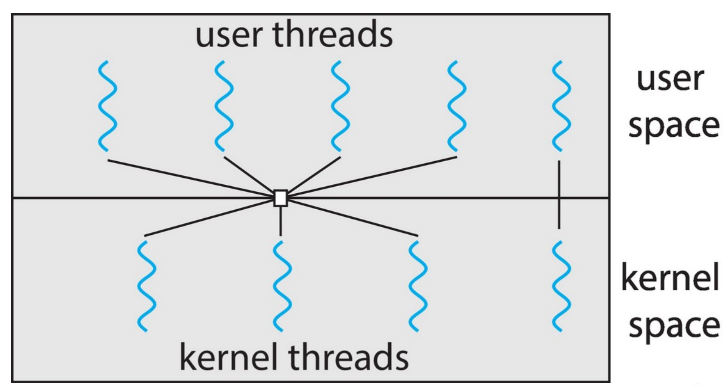


Threading Models

Many-to-Many

- maps N user threads into M <= N kernel threads
  - value o M may vary depending on the application, num. CPU cores
- (+) more flexible than previous models
  - applications spawn N threads as they see fit ...
  - … knowing that at least M will be scheduled by the kernel …
  - … and that any call to a blocking syscall will not block the process
  - variant: **2-level** model (allows One-to-One to some threads)
- (-): harder to implement; not as useful with so many cores in modern CPUs





2.9 Threading Libraries

Threading Libraries

- **threading library:** exposes an API to create and manage threads
  - **user-level library**: code and data-structures in user-space; calling an API function implies calling a function implemented in user-space
  - **kernel-level library**: code and data-structures in the kernel; calling an API function implies calling a syscall (executed in kernel mode)
  - main explicit threading libraries: POSIX, Windows and Java threads

- **Synchronous vs Asynchronous threading**
  - **synchronous threading**: right after the creation of a “child” thread, the “parent” thread awaits for the “child” termination before continuing (if N “children” were created, the “parent” will wait for all to terminate)
  - **assynchronous threading**: right after the creation of a “child” thread, the “parent” continues its execution, without awaiting for the “child” to terminate (if N “children” were created, the “parent” will await for none)

Explicit Threading Libraries

- **POSIX threads (Pthreads)**
  - POSIX standard (IEEE 1003.1c) which specifies a threading API
  - the threading API by default in UNIX, Linux and MacOS systems
  - user-level and kernel-level implementations available
  - global data (outside functions) is visible and shared by all threads
- **Windows threads**
  - provided by a kernel-level library
  - global data (outside functions) is visible and shared by all threads
  - API and workflow similar to Pthreads
- **Java threads**
  - provided by a user-level library
  - implemented based on the default threading API of the OS
  - Java does not have native global data and so the sharing of data between threads must be explicitly programmed

Explicit Threading Libraries

- **POSIX threads (example)**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_WORKERS 5

long GLOBAL_squares[NUM_WORKERS];

void *aThread(void *arg)
{
 long tid=(long)arg;

 GLOBAL_squares[tid]=tid*tid;
 printf("thread %ld: square = %lu\n", tid, GLOBAL_squares[tid]);
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t threads[NUM_WORKERS];
 long t, sum=0;

 for(t=0;t<NUM_WORKERS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 for(t=0; t<NUM_WORKERS; t++) {
     pthread_join(threads[t], NULL);
     printf("thread main: joined with thread %ld\n", t);
     sum += GLOBAL_squares[t];
 }

 printf("thread main: sum = %lu: exiting\n", sum);
 pthread_exit(NULL);
}
```

```
thread main: creating thread 0
thread main: creating thread 1
thread 0: square = 0
thread 1: square = 1
thread main: creating thread 2
thread main: creating thread 3
thread 2: square = 4
thread main: creating thread 4
thread 3: square = 9
thread main: joined with thread 0
thread main: joined with thread 1
thread main: joined with thread 2
thread main: joined with thread 3
thread 4: square = 16
thread main: joined with thread 4
thread main: sum = 30: exiting
```

(compile with
`gcc pthreads-example.c -o pthreads-example.exe -lpthread`)

Appendix – Extra Topics

Examples of IPC Systems – SyS V

- UNIX System V Message Queues: **sender** example

```
#define MAXSIZE 128
main() {
  int msqid; key_t key = 0x12345678; size_t bufferlen;
  struct { long mtype; char mtext[MAXSIZE]; } buffer;

  msqid = msgget(key, IPC_CREAT | 0666);

  buffer.mtype = 1;
  strcpy(buffer.mtext, "first message");
  bufferlen = strlen(buffer.mtext) + 1 ;
  msgsnd(msqid, &buffer, bufferlen, IPC_NOWAIT);

  buffer.mtype = 1;
  strcpy(buffer.mtext, "second message");
  bufferlen = strlen(buffer.mtext) + 1 ;
  msgsnd(msqid, &buffer, bufferlen, IPC_NOWAIT);
}
```

synchronization issues:
with IPC_NOWAIT the sending is non-blocking (asynchronous)

Examples of IPC Systems – SyS V

- UNIX System V Message Queues: **receiver** example

```
#define MAXSIZE 128
main() {
   int msqid; key_t key = 0x12345678; long type = 1;
   struct { long mtype; char mtext[MAXSIZE]; } buffer;

   msqid = msgget(key, 0666);

   msgrcv(msqid, &buffer, MAXSIZE, type, 0);
   printf("%s\n", buffer.mtext);

   msgrcv(msqid, &buffer, MAXSIZE, type, 0);
   printf("%s\n", buffer.mtext);

   msgctl(msqid, IPC_RMID, NULL);
}
```

synchronization issues:
by default, reception is blocking (synchronous)

Theoretical Unit 2

References:

- "Operating System Concepts, 10th Ed.", Silberschatz \& Galvin, Addison-Wesley, 2018: Chapters 3 and 4

- Anatomy of a Program in Memory:
https://web.archive.org/web/20180206141815/https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

- Anatomy of Linux Process Management:
https://developer.ibm.com/tutorials/l-linux-process-management/

End of Theoretical Unit 2