Principles of Operating Systems

Lecture 2 - Processes and Threads

Ardalan Amiri Sani (ardalan@uci.edu)

[*lecture slides contains some content adapted from : previous slides by Prof. Nalini Venkatasubramanian, and course text slides © Silberschatz*]

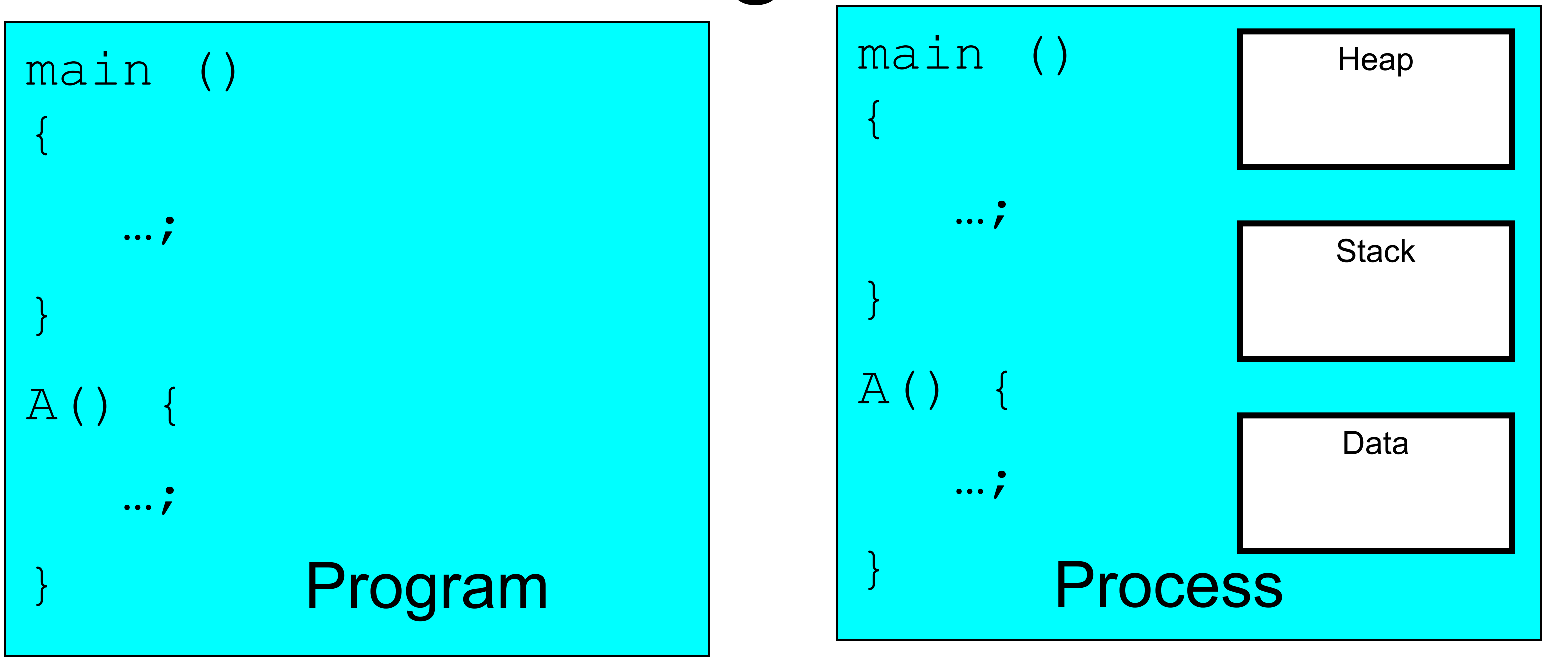
Outline

- Processes
- Threads
- Interprocess Communication

Process Concept

- An operating system executes a variety of programs
- Process - an instance of a program in execution (with limited rights)
  - For now, we assume that the process has a single thread of execution. Therefore, the process execution proceeds in a sequential fashion
- A process address space contains
  - Stack, heap, data and code sections

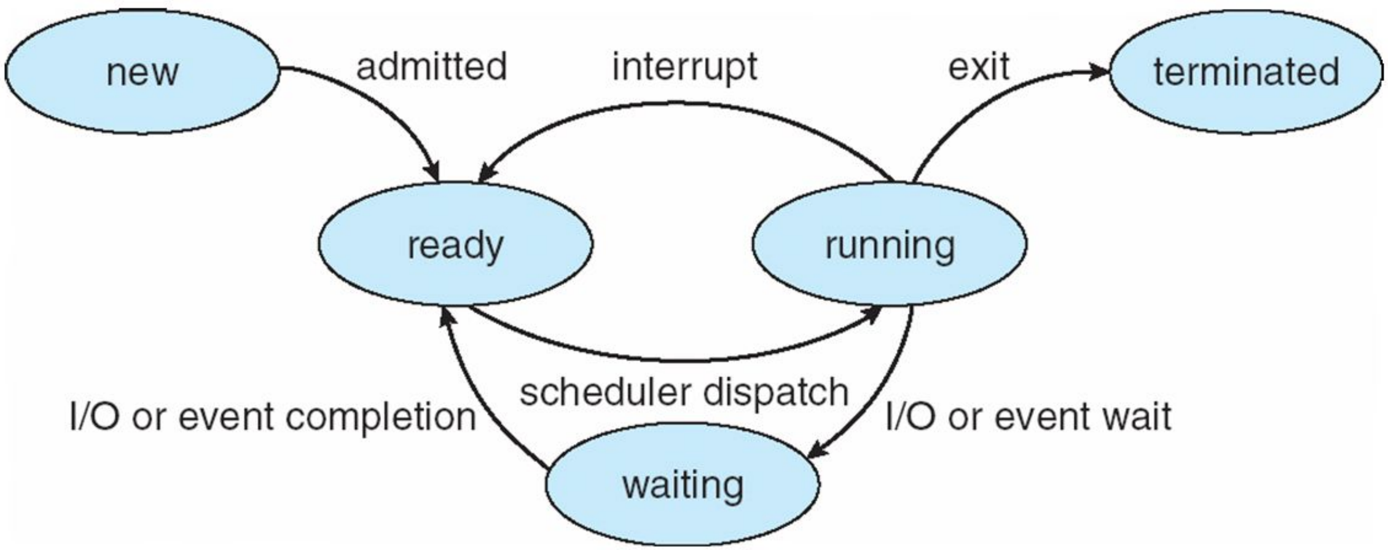
Process =? Program



- A process is one instance of a program in execution
- I run Vim on lectures.txt, you run it on homework.java – Same program, different processes
- A program can invoke more than one process
  - A web browser launches multiple processes, e.g., one per tab

Process States

- A process changes state as it executes.

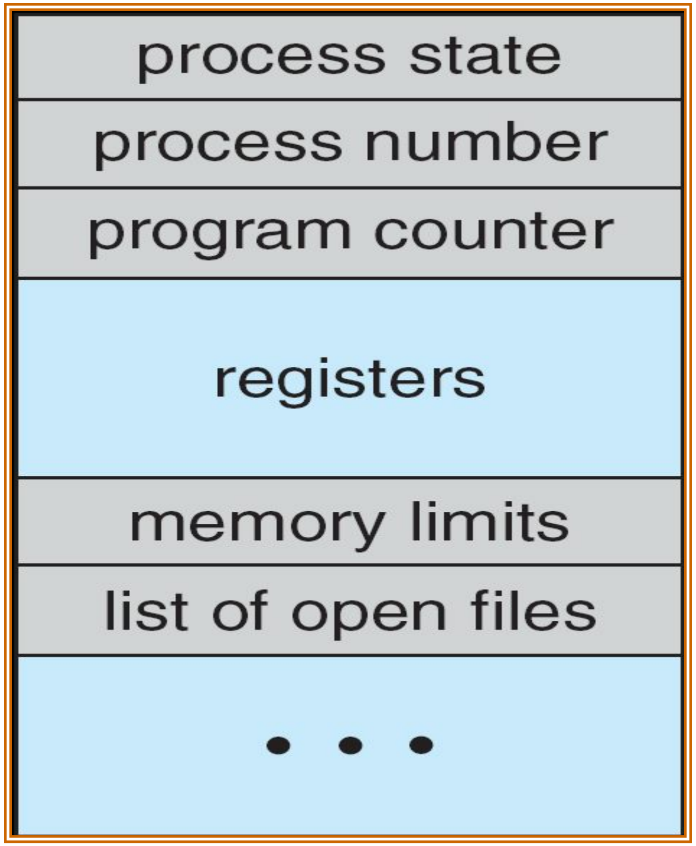


Process States

- New - The process is being created.
- Running - Instructions are being executed.
- Waiting - Waiting for some event to occur.
- Ready - Waiting to be assigned to a processor.
- Terminated - Process has finished execution.

Process Control Block

- Kernel maintains a PCB for each process
- Contains information associated with each process
  - 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
  - I/O status information – list of open files

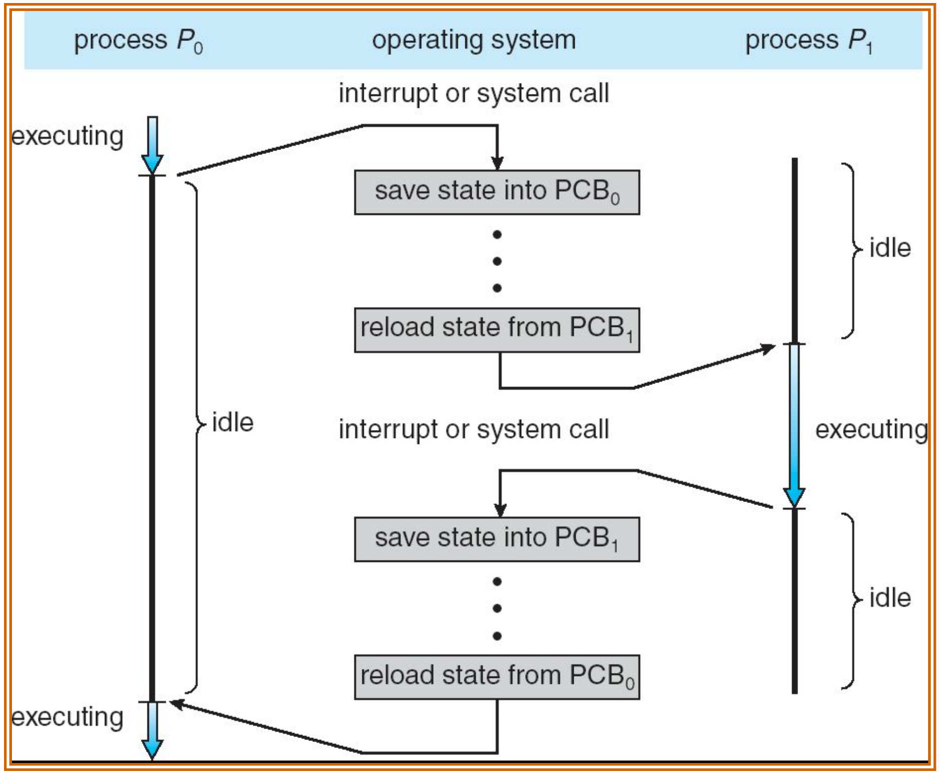


Process Control Block

Enabling Concurrency: Context Switch

- Operation that switches CPU from one process to another process
  - the CPU must save the state of the old process into its PCB and load the state of the new process from its PCB.
- Context-switch time is overhead
  - System does no useful work while switching
  - Overhead sets minimum practical switching time; can become a bottleneck
- Time for context switch is dependent on hardware support (typically 1- 1000 microseconds).

CPU Switch From Process to Process

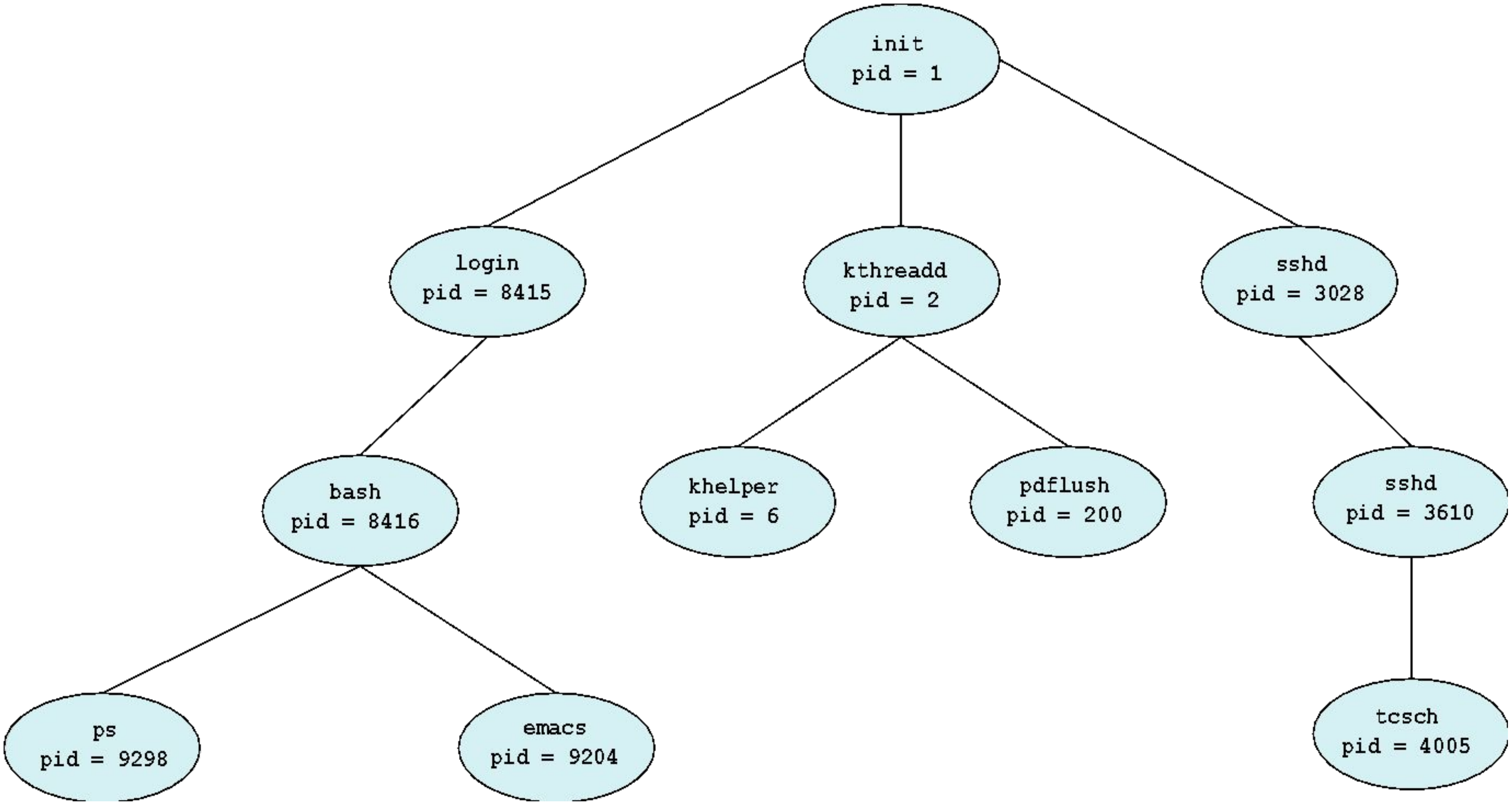


- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
- The scheduler decides which process to execute next (scheduler will be discussed in the next lecture)

Process Creation

- Processes are created by other processes
  - The kernel implements the mechanism to create a new process in the form of a syscall.
- Process which creates another process is called a *parent* process; the created process is called a *child* process.
- Result is a tree of processes

A tree of processes in Linux



Fun question: who creates the init process?

Fun question: who creates the init process?

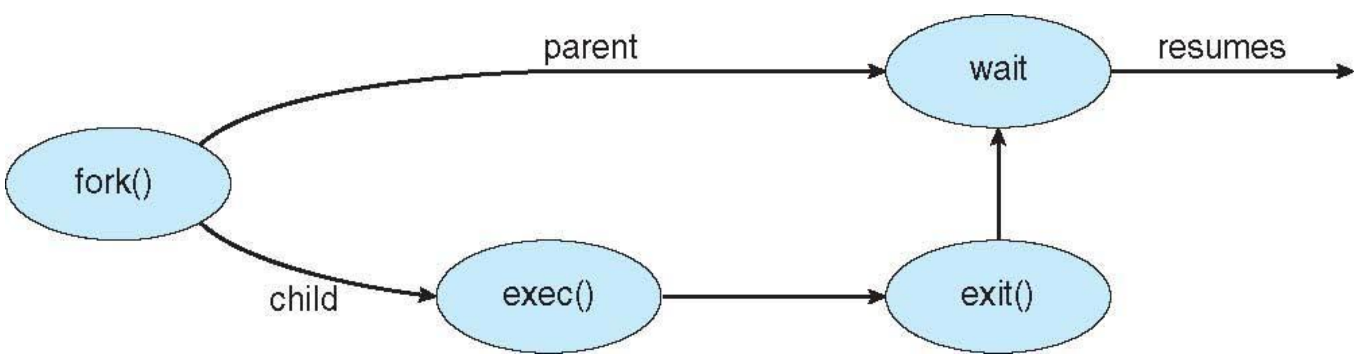
- Kernel, all on its own.

What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up the address space (e.g., set up new page tables for address space)
  - More expensive
- Copy data from parent process? (Unix fork())
  - Semantics of Unix fork() are that the child process gets a complete copy of the parent memory
  - Originally *very* expensive
  - Much less expensive with “copy on write”
- Copy I/O state (file handles, etc)
  - Medium expense

UNIX Process Creation

- Address space
  - First, child's address space is duplicate of parent's
  - Then, child *can* load a new program
- Fork system call creates new processes
- exec() system call is used after a fork to replace the processes memory space with a new program.



Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded a threshold for allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and wants to terminate the child process too

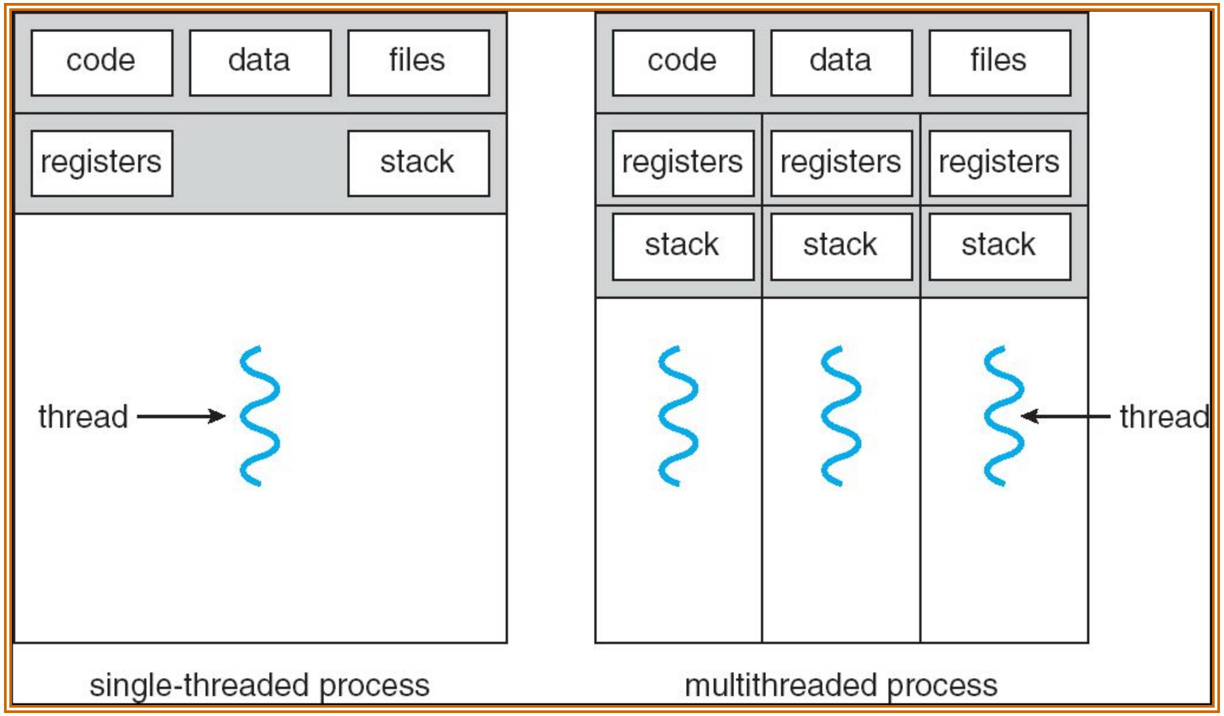
Process Termination

- Zombie process: a child process that has terminated, but its parent hasn’t called wait() yet.
- Orphan process: a child process, whose parent process has died. Orphan process is adopted by the init process.

Threads

- Processes do not share resources well and they have high context switching overhead
- Idea: Separate concurrency from protection
- Multithreading: *a single program made up of a number of different concurrent activities*
- A thread
    - basic unit of CPU execution; it has separate:
        - program counter, register set, and stack space
  - A thread shares the following with peer threads:
        - memory address space including code section, data section, heap, etc. (Q. can one thread access another thread's stack?)
        - OS resources (open files)
        - No protection between threads

Single and Multithreaded Processes



- Threads encapsulate execution and concurrency
- Process encapsulates protection

Threads (Cont.)

- In a multi-threaded process, while one thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in the same job results in higher throughput and improved performance.

Thread State

- State shared by all threads in the process
  - Content of memory (global variables, heap)
  - I/O state (open files, network connections, etc.)
- State “private” to each thread
  - Kept in TCB = Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack
  - Thread (execution) state -
    - *new, ready, waiting, running, terminated*

Threads (cont.)

- Switching between two threads in the same process still requires a register set switch, but no memory management related work!

- Only one thread can run on a CPU at a time.

Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads

Kernel Threads

- Supported by the Kernel
  - Threads created and managed directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode for scheduling
- Example
  - Linux

User Threads

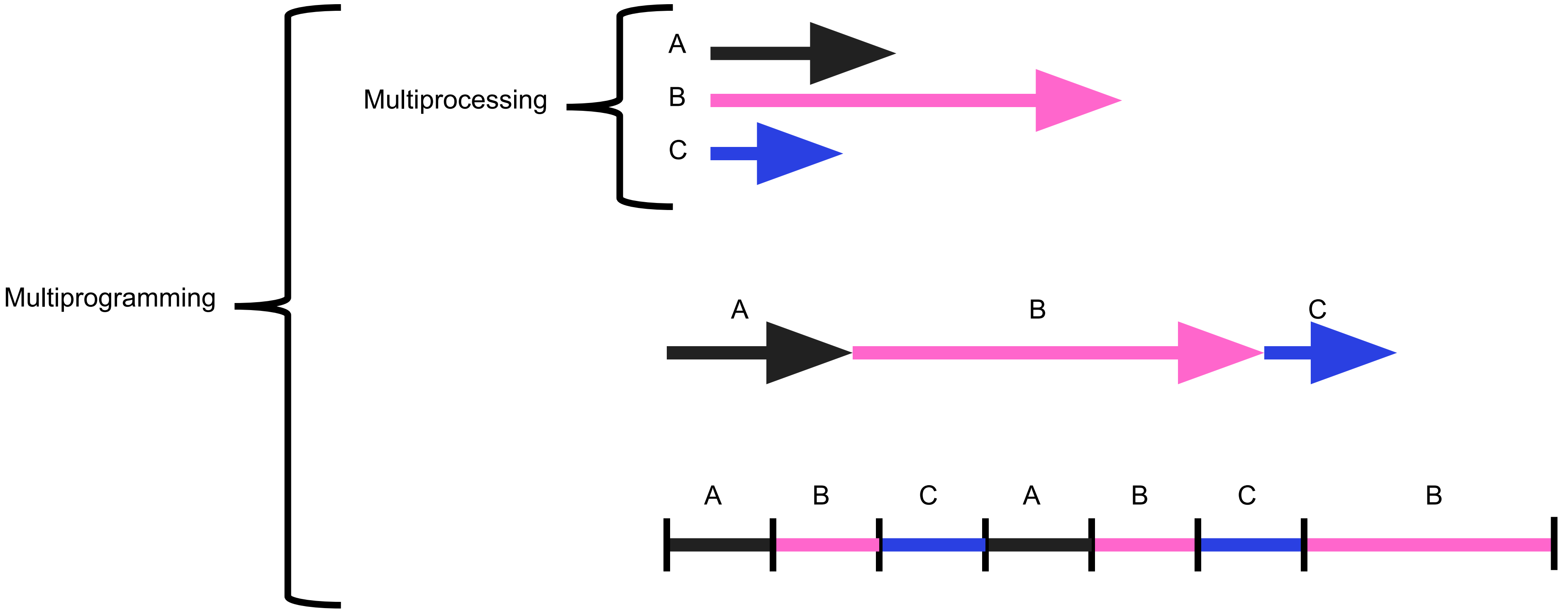
- Supported above the kernel, via a set of library calls at the user level.
  - Thread management done by user-level threads library
    - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Advantages
    - Cheap, Fast
      - Threads do not need to cross to the kernel for scheduling
  - Disadv: Threads will not run in parallel, only one thread at a time per kernel thread
- Example thread libraries:
  - POSIX Pthreads can support user threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
    1. Signal is generated by a particular event
    2. Signal is delivered to a process
    3. Signal is handled by one of two signal handlers:
        1. default
        2. user-defined
- Every signal has **default handler** that runs when handling signal
    - **User-defined signal handler** can override default
        - Can’t override SIGKILL and SIGSTOP

Multi (processing, programming, threading)

- Definitions:
  - Multiprocessing: Multiple processors/CPUs
  - Multiprogramming: Multiple jobs/processes
  - Multithreading: Multiple threads per process



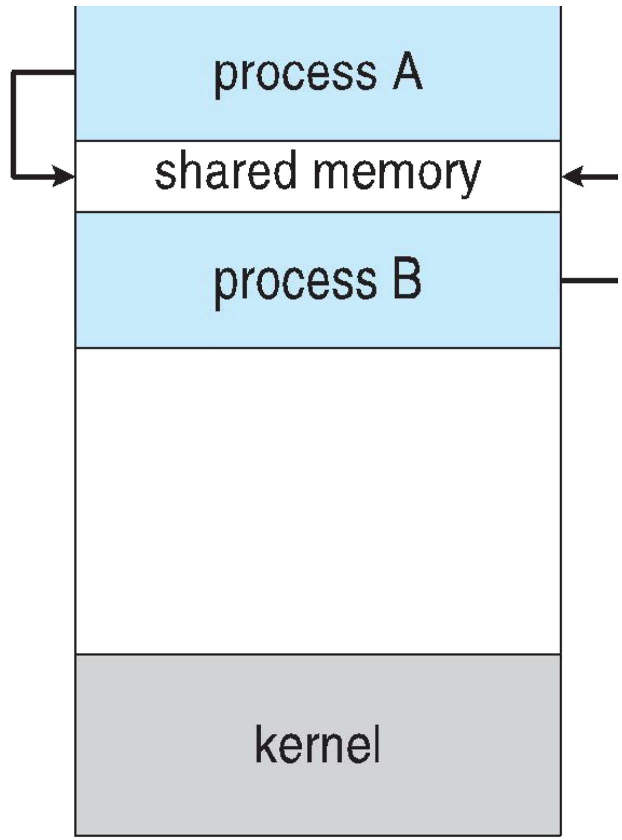
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need to communicate and share data. For this purpose, they use **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the 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 will be discussed in future lectures.

Interprocess Communication – Shared Memory



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **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 BUFFER_SIZE 10
typedef struct {
  . . .
} item;

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

Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

```
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 */
}
```

Bounded-Buffer – Shared-Memory Solution

- How many elements can be stored in the buffer at most at a given time?

```
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;
}
```

Producer

```
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 */
}
```

Consumer

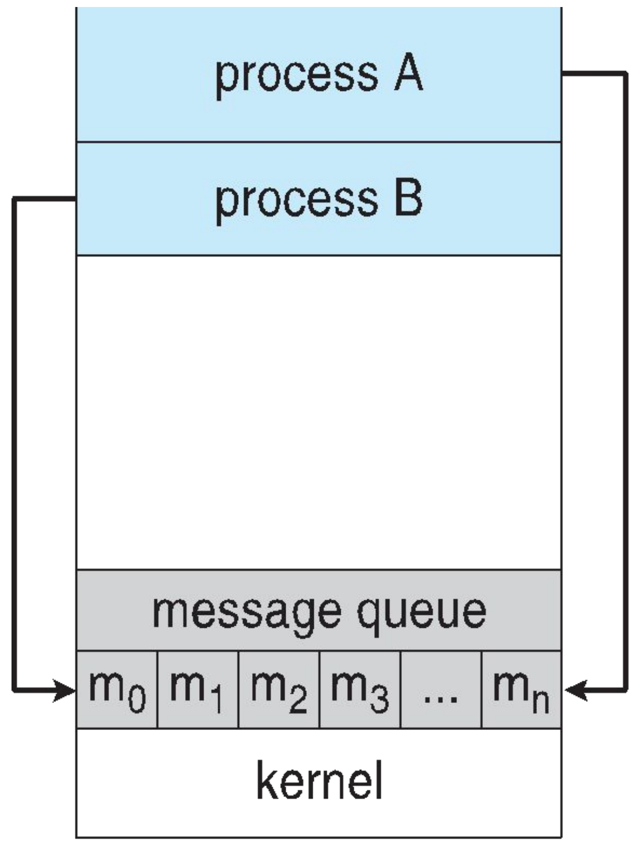
Bounded-Buffer – Shared-Memory Solution

- Can only use BUFFER_SIZE-1 elements

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – 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

Interprocess Communication – Message Passing



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?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Main memory (Figure in slide 38)
    - Hardware bus
    - Network

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
  - Link may be unidirectional or 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

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues
  - **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 (Cont.)

- Producer-consumer becomes trivial

Producer

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}
```

Consumer

```
message next_consumed;
while (true) {
   receive(next_consumed);

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

Message passing (Cont.)

- **Q. What are the send and receive here? Blocking or non-blocking?**

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}
```

Producer

```
message next_consumed;
while (true) {
   receive(next_consumed);

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

Consumer

Buffering

- Queue of messages attached to the link is 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

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ```
  - Also used (without the O_CREAT flag) to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```

  - Now the process could write to the shared memory

    ```
    sprintf(shared_memory_addr, "Writing to shared
    memory");
    ```

IPC POSIX Producer (no synchronization)

```
#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 (no synchronization)

```
#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;
}
```

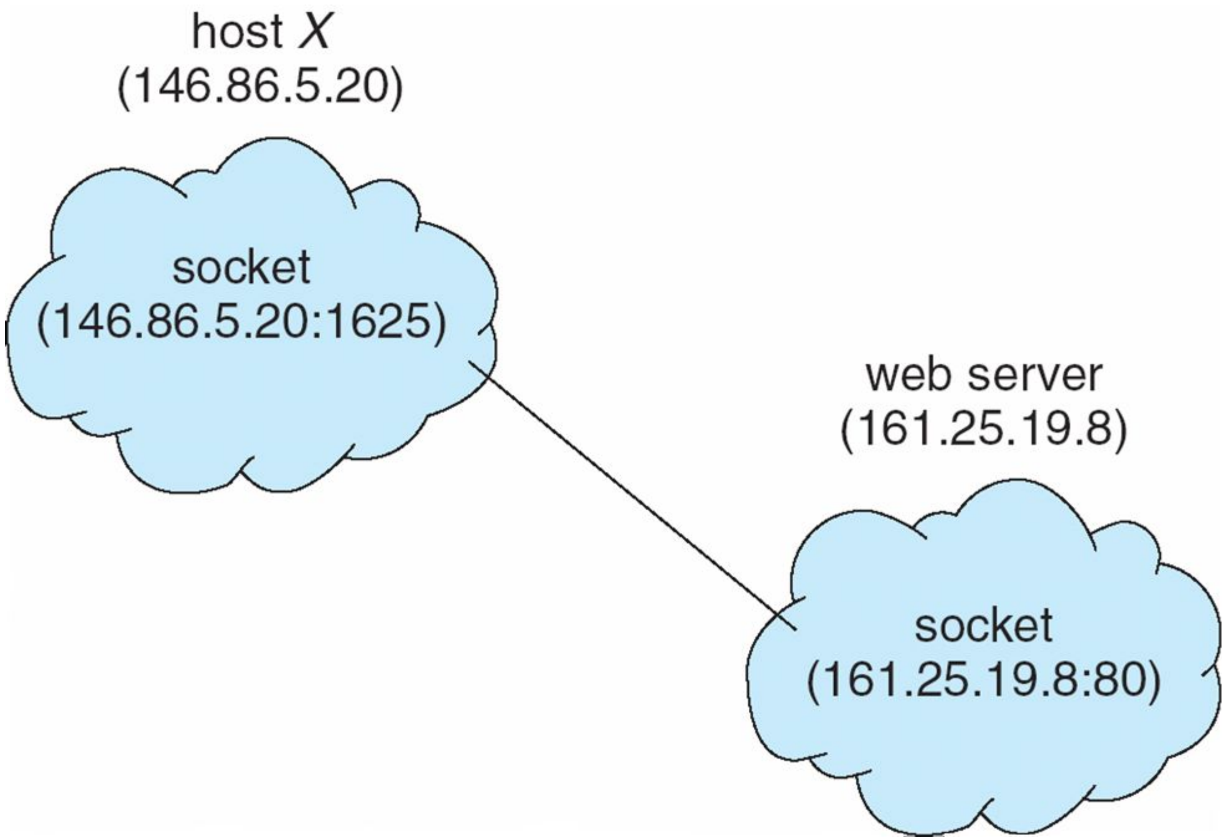
Other IPC solutions

- Sockets
- Remote Procedure Calls
- Pipes

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
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



Sockets example

```
int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in *serv_addr;
    char buffer[256];

    portno = ...;
    server_addr = …;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");

    if (connect(sockfd, serv_addr, sizeof(*serv_addr)) < 0)
        error("ERROR connecting");

    /* Here, fill up the buffer with the message to send */

    n = write(sockfd, buffer, strlen(buffer));
    if (n < 0)
         error("ERROR writing to socket");

    /* Here, empty the buffer */

    n = read(sockfd, buffer, 255);
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    close(sockfd);
    return 0;
}
```

based on:
http://www.linuxhowtos.org/data/6/client.c

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems

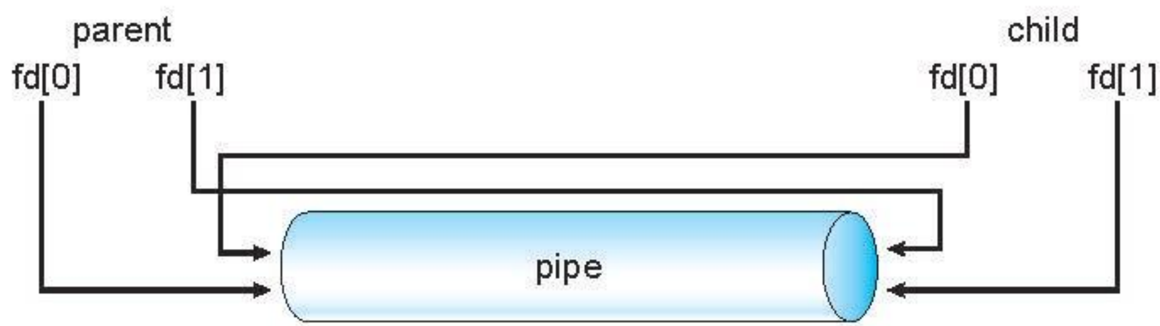
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

Pipes

- Acts as a conduit allowing two processes to communicate
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- 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
- Require parent-child relationship between communicating processes



Ordinary Pipes

(see full example in the book)

```
#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;

     if (pipe(fd) == -1) {
          /* handle error */
     }

     pid = fork();

     if (pid < 0) {
          /* handle error */
     }

     If (pid > 0) { /* parent process */
          close(fd[READ_END]);
          write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
          close(fd[WRITE_END]);
     } else { /* child process */
          close(fd[WRITE_END]);
          read(fd[READ_END], read_msg, BUFFER_SIZE);
          printf(“read %s”, read_msg);
          close(fd[READ_END]);
     }
     return 0;
}
```

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