"People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird."

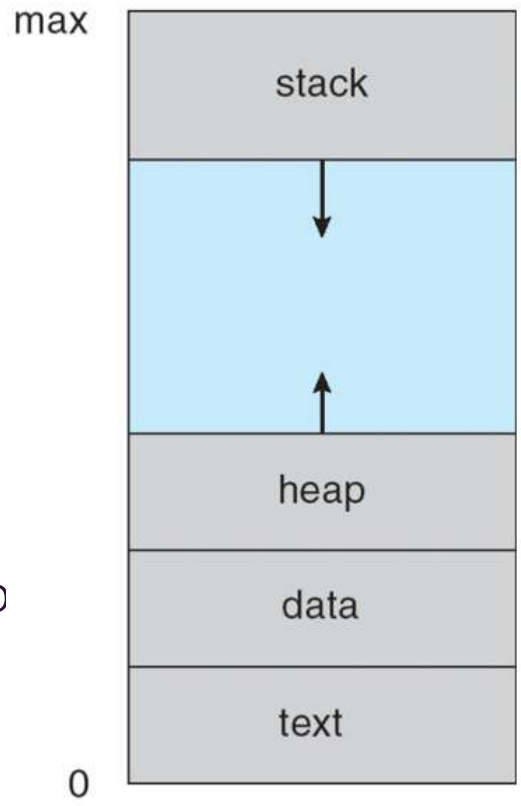
- D. Knuth

Process Management

These slides were compiled from the OSC textbook slides (Silberschatz, Galvin, and Gagne) and the instructor's class materials.

Process Concept

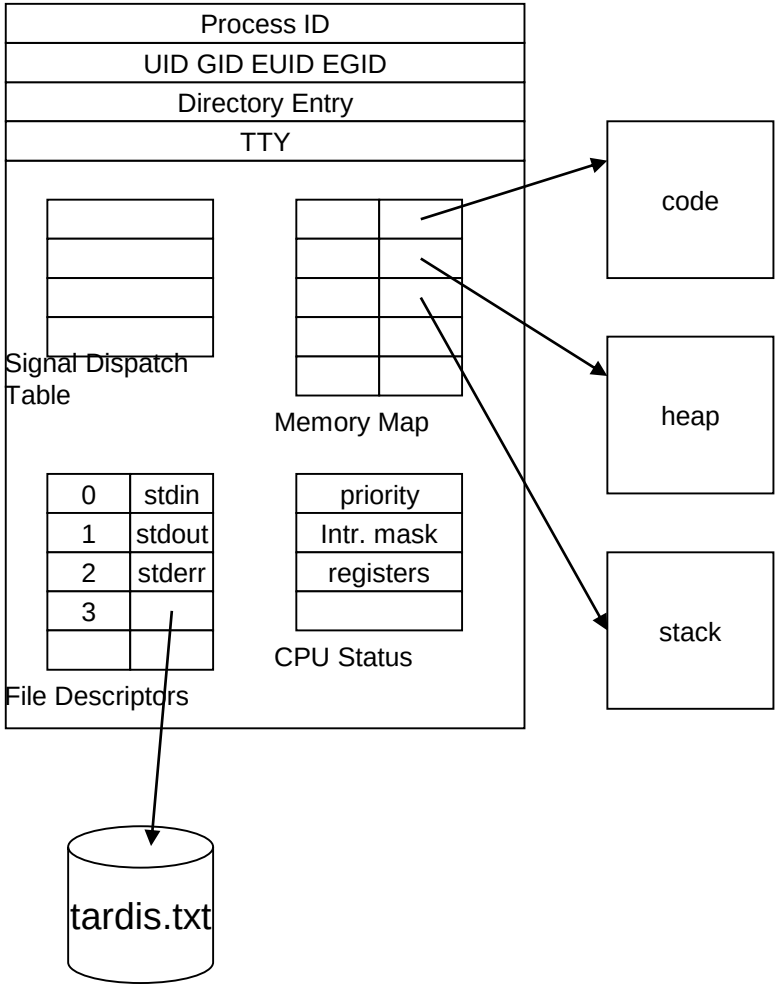
- **Process** – a program in **execution**; process execution must progress in **sequential** fashion.
- Textbook uses the terms *job* and *process* almost interchangeably.
- A process includes:
  - **Program counter**
  - **Stack** (local variables)
  - **Data section** (global data)
  - **Text** (code)
  - **Heap** (dynamic data)
  - **Files** (cin, cout, cerr, other file descripto



Process Control Block (PCB)

Contains Information associated with each process, such as…

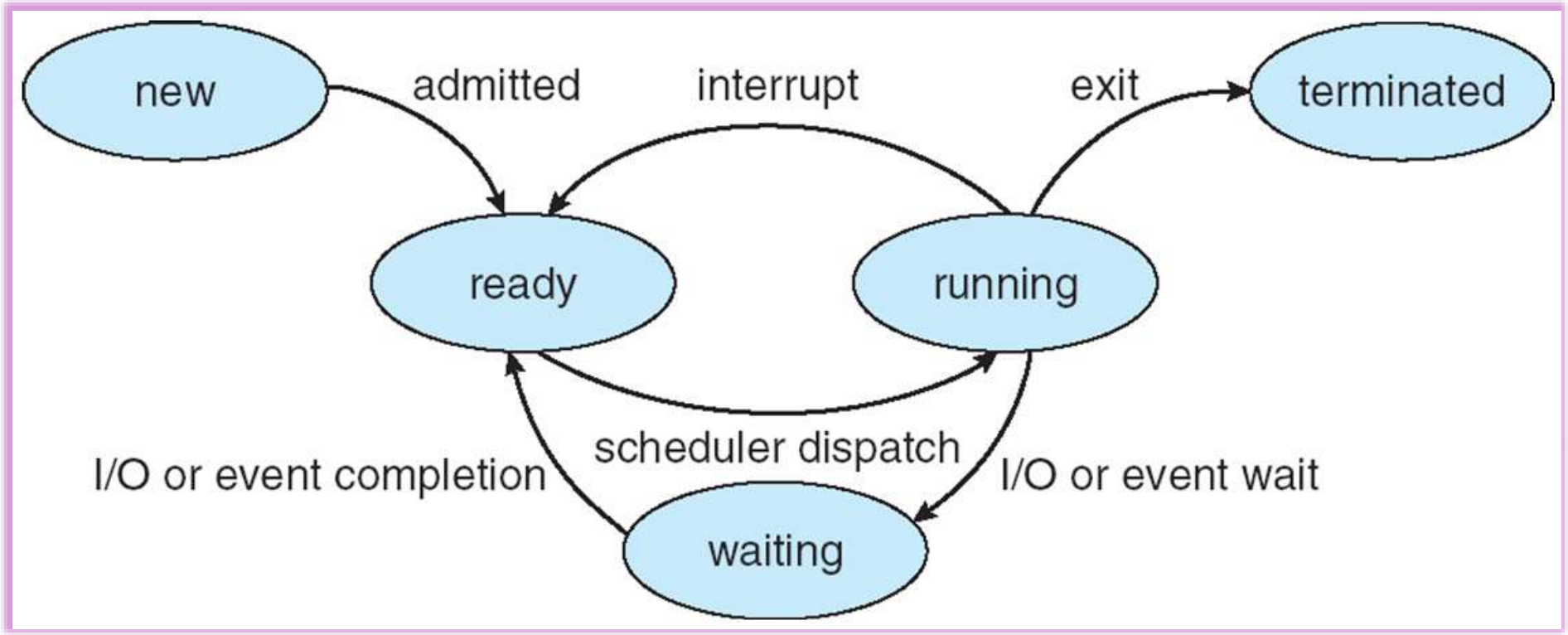
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



This is an overview, we'll discuss various parts in more detail in upcoming chapters

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

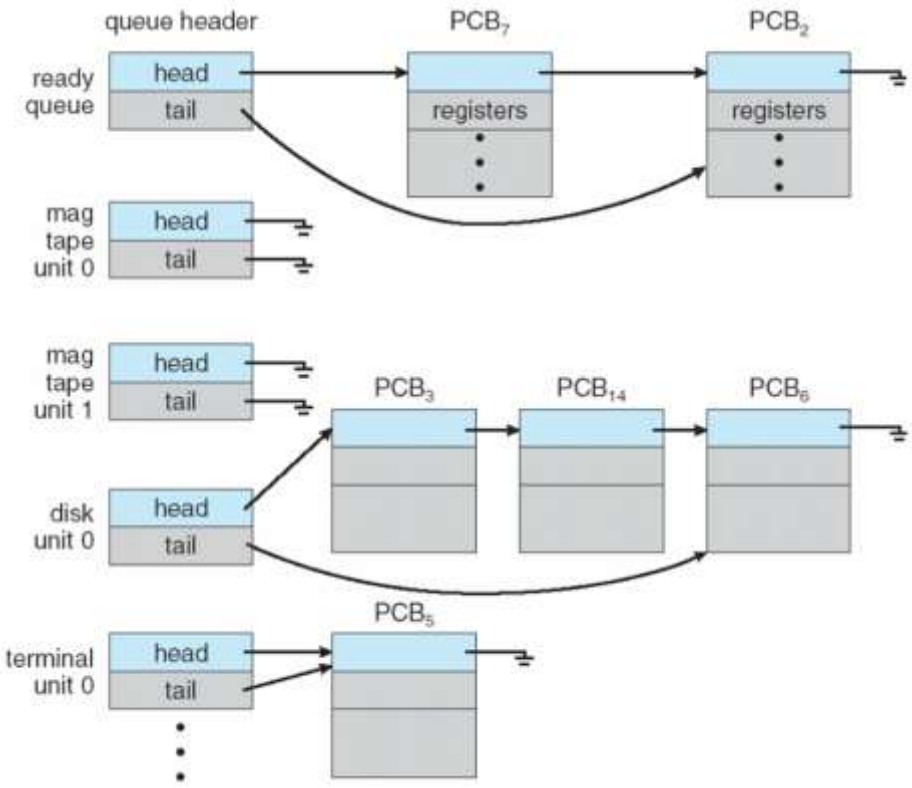


Context Switch: Changing processes

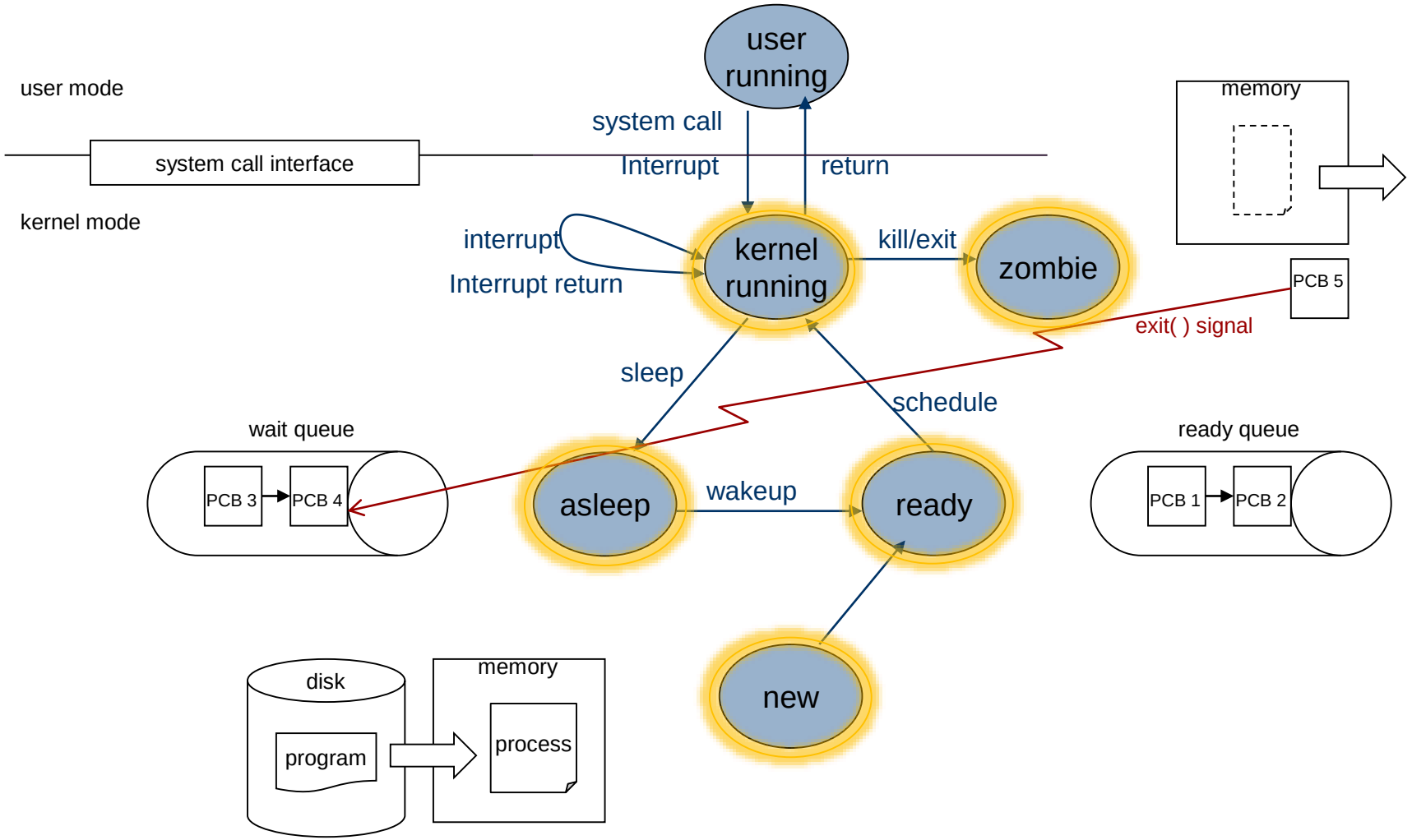
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while switching
- Time dependent on hardware support

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



Process Status



Process vs. Thread

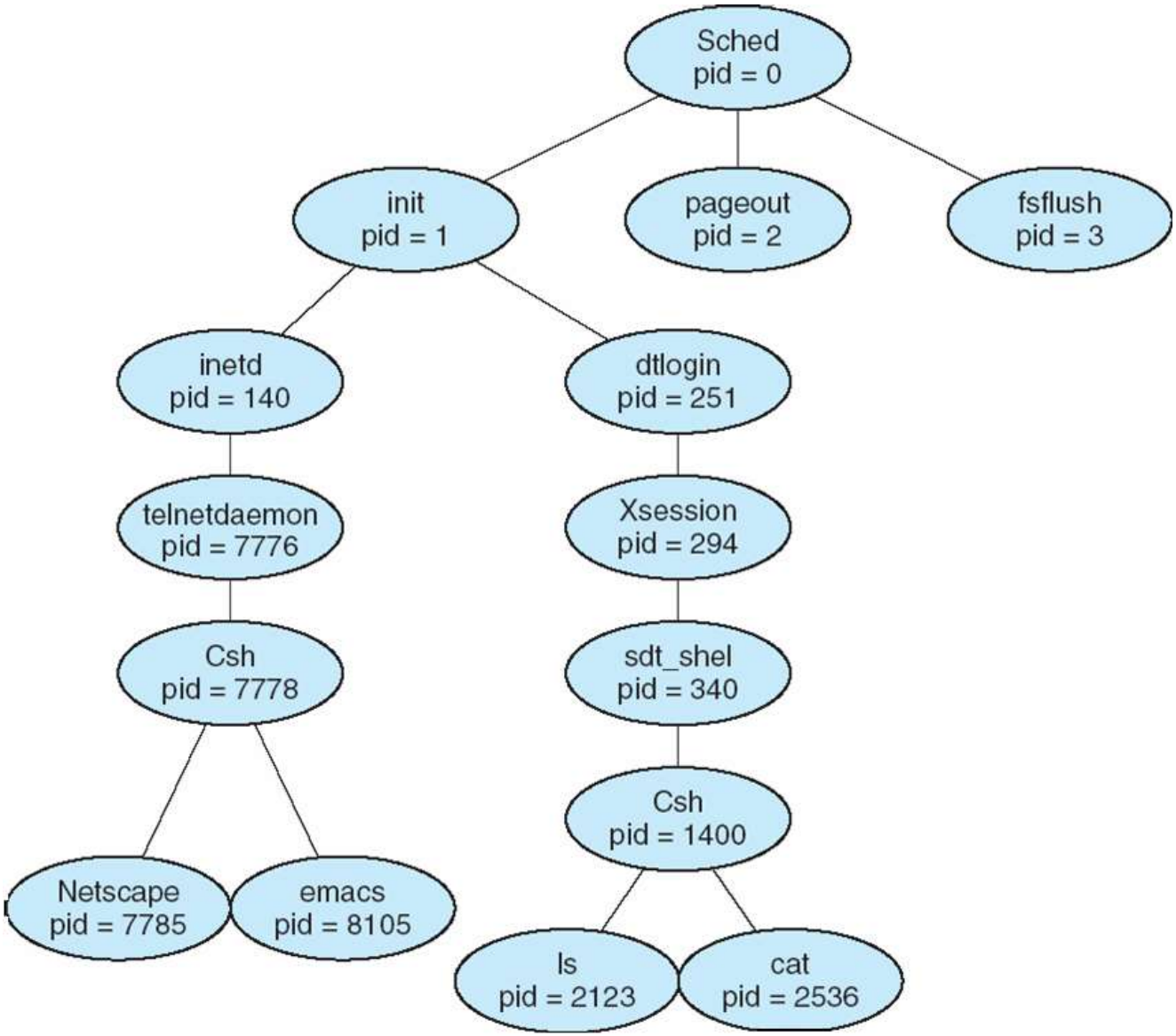
- Thread is a "lightweight process"
- Process consists of CPU state:
  - registers, memory, OS info (open files, PID, etc.),
  - in a thread system there is a larger entity called **task**
- A **task** (or **process**) consists of memory, OS info and threads
  - Each thread is a unit of execution, consisting of a stack and CPU state
  - Multiple threads resemble multiple processes, but multiple thread use the same code, globals and heap
- Processes can communicate through the OS
- Threads can communicate through memory, appearing like each thread executes on its own CPU and all threads share the same memory

Process Creation

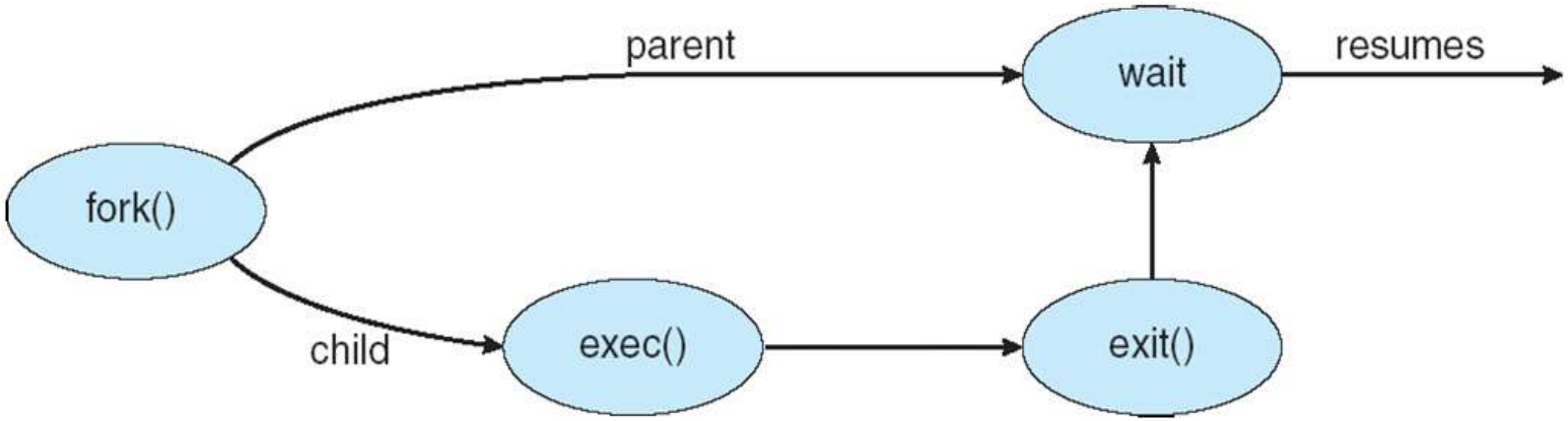
- Parent process creates children processes.
- Resource sharing
  - Resource inherited by children: file descriptors, shared memory and system queues
  - Resource not inherited by children: address space
- Execution
  - Parent and children execute concurrently.
  - Parent waits by **wait** system call until children terminate.
- LINUX examples
  - **fork** system call creates new process.
  - **execlp** system call used after a **fork** to replace the process' memory space with a new program.

- Note for CSS430: ThreadOS-unique system calls: SysLib.exec and Syslib.join

A Tree of Processes On A Typical UNIX



Process Creation



Try
ping -c15 localhost & ls

C Program Forking Separate Process

Note, this fork() returns child pid

```c
#include <stdio.h>   //for printf
#include <stdlib.h>  //for exit
#include <unistd.h>  //for fork, exe
int main(int argc, char *argv[])
{
   int pid;  //process ID
   //fork another process
   pid = fork();
   if (pid < 0) {  //error occurred
      fprintf(stderr, "Fork Failed");
      exit(EXIT_FAILURE);
   }
   //---------- CHILD SECTION ------
   else if (pid == 0) {
      execlp("/bin/ls","ls","-l",NULL
   }
   //---------- PARENT SECTION -----
   else {
      //parent will wait for the chi
      wait(NULL);
      printf("Child Complete");
      exit(EXIT_SUCCESS);
   }
}
```

Parent Process

Child Process

```c
#include <stdio.h>   //for printf
#include <stdlib.h>  //for exit
#include <unistd.h>  //for fork, execlp
int main(int argc, char *argv[])
{
   int pid;  //process ID
   //fork another process
   pid = fork();
   if (pid < 0) {  //error occurred
      fprintf(stderr, "Fork Failed");
      exit(EXIT_FAILURE);
   }
   //---------- CHILD SECTION ----------
   else if (pid == 0) {
      execlp("/bin/ls","ls","-l",NULL);
   }
   //---------- PARENT SECTION ----------
   else {
      //parent will wait for the child to complete
      wait(NULL);
      printf("Child Complete");
      exit(EXIT_SUCCESS);
   }
}
```

Another view of Process Creation

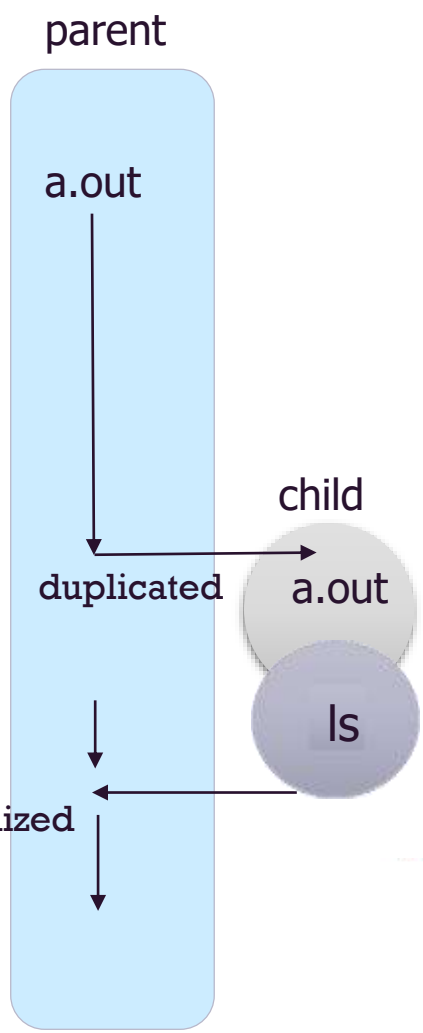
Parent Process

```
#include<stdio.h>///for printf
#include<stdlib.h
#include<unistd.h
int main(int argc,
{
    int pid;//pro
    //fork another
    pid=fork();
    if(pid<0){
        fprintf(stde
        exit(EXIT_FA
    }
    //----------C
    else if(pid==
        execlp("/bi
    }
    //----------P
    else{
        //parent wa
        wait(NULL);
        printf("Chi
        exit(EXIT_SU
    }
}
```

Child Process

```
. . .
#define PROGRAM_NAME (ls_mode == LS_LS ? "ls" \
                      : (ls_mode == LS_MULTI_COL \
                         ? "dir" : "vdir"))
int
main (int argc, char **argv)
{
  int i;
  struct pending *thispend;
  int n_files;
. . .
 if (print_with_color)
    {
      int j;

      if (used_color)
        {
          /* Skip the restore when it would be a no-op, i.e.,
             when left is "\033[" and right is "m".  */
          if (!(color_indicator[C_LEFT].len == 2
            && memcmp (color_indicator[C_LEFT].string, "\033[", 2) == 0
            && color_indicator[C_RIGHT].len == 1
            && color_indicator[C_RIGHT].string[0] == 'm'))
           restore_default_color ();
        }
fflush (stdout);
. . .
Exit status:\n\
 0  if OK,\n\
 1  if minor problems (e.g., cannot access subdirectory),\n\
 2  if serious trouble (e.g., cannot access command-line argument).\n\
"), stdout);
      emit_ancillary_info ();
    }
  exit (status);
}
```

`ls` code (Child Process)



Process Termination

- Process termination occurs when
  - It executes the **last** statement
  - It executes **exit** system call explicitly 

- Upon process termination
  - Termination code is passed from child (via **exit**) to parent (via **wait**)
  - Process' resources are de-allocated by OS

- Parent may terminate execution of children processes (via **kill**) when
  - Child has *exceeded* allocated resources
  - Task assigned to child is *no longer required*
  - Parent is exiting (cascading termination)
    - Some operating system does not allow child to continue if its parent terminates

Inter Process Communication

Cooperating Processes

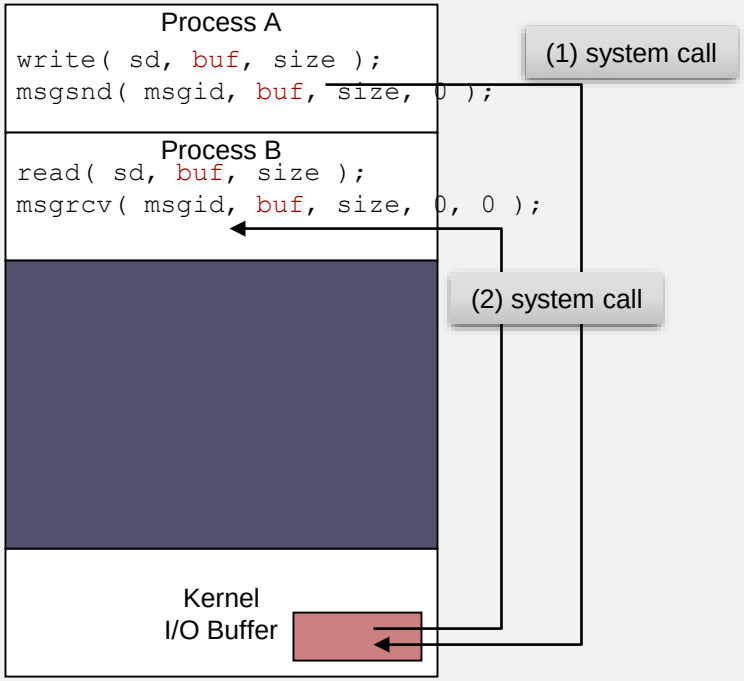
- *Process independence*: Processes belonging to a ***different user*** do not affect each other unless they give each other some access permissions

- *Process Cooperation*: Processes spawned from the same user process share some resources and communicate with each other through them (e.g., shared memory, message queues, pipes, and files)

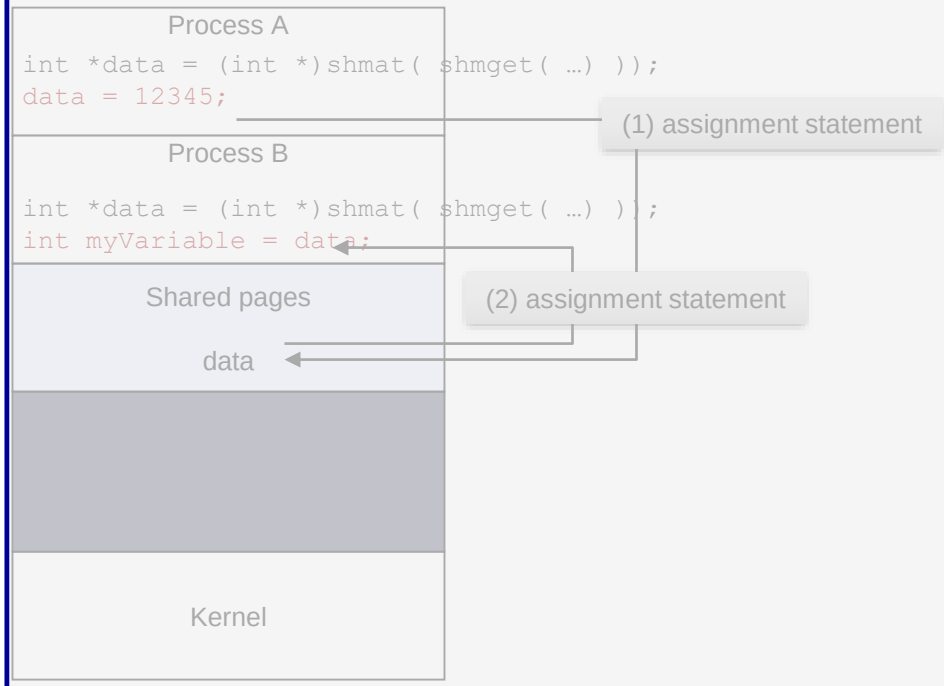
- Uses of process cooperation
  - Information sharing: (sharing files)
  - Computation speed-up: (parallel programming)
  - Modularity: (like `who | wc -l`, one process lists current users and another counts the number of users.)
  - Convenience: (net-surfing while working on programming with emacs and g++)

Communication Models

Message passing



Shared memory



Message Passing

- Message system – processes communicate with each other without resorting to shared variables.

- Inter-Process Communication facility 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
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Message Passing

send (*P, message*) – send a message to process P

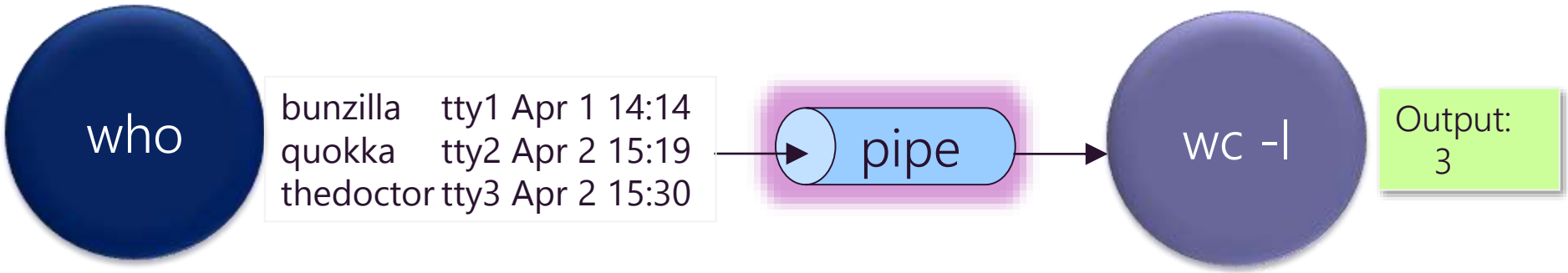
receive(*Q, message*) – receive a message from process Q

- How can a process locate its partner to communicate with?
  - Problem: Processes are created and terminated dynamically, and thus, a process' partner may be gone
  - Direct communication takes place between a parent and its child process in many cases.

    Example: pipe

Pipes (Direct communication)

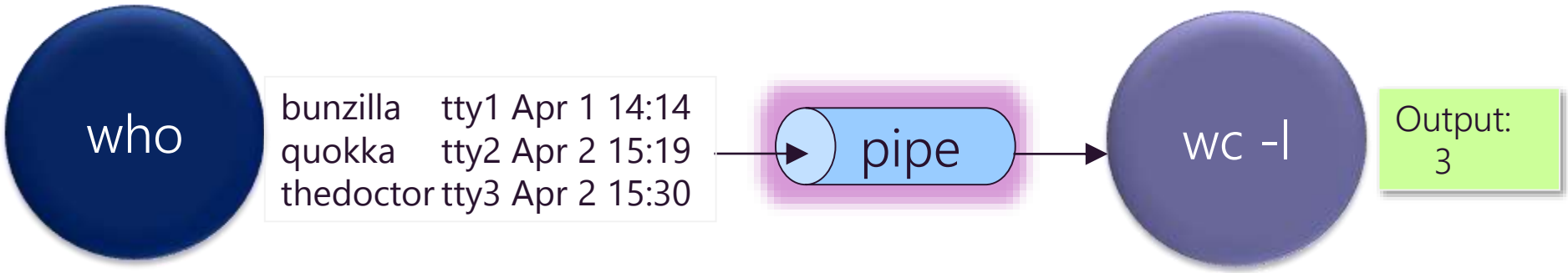
```
who  |  wc -l
```



- A **`pipe`** is a system call used for inter-process communication
- A **`pipe`** connects between two processes such that the *standard output* from one process becomes the *standard input* of the other process

Pipes: A Producer-Consumer Problem

```
who | wc -l
```



- Producer process:
  - **who** produces a list of current users.
- Consumer process
  - **wc** receives it for counting #users.
- Communication link:
  - OS provides a **pipe**.

Child sends information to the parent process

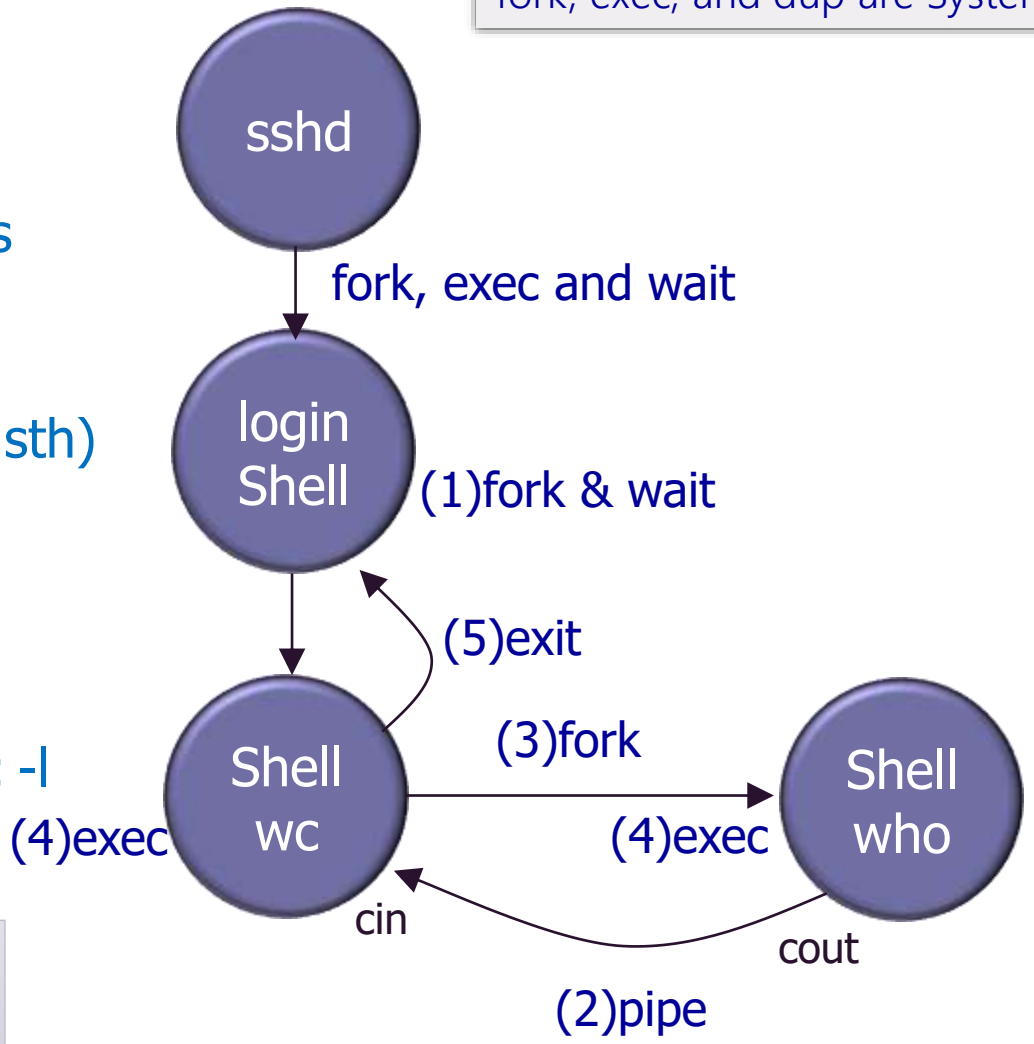
Linux Shell

fork, exec, and dup are System calls

mirkwood login: daenerys

mirkwood[1]% (you type sth)

mirkwood[1]% who | wc -l



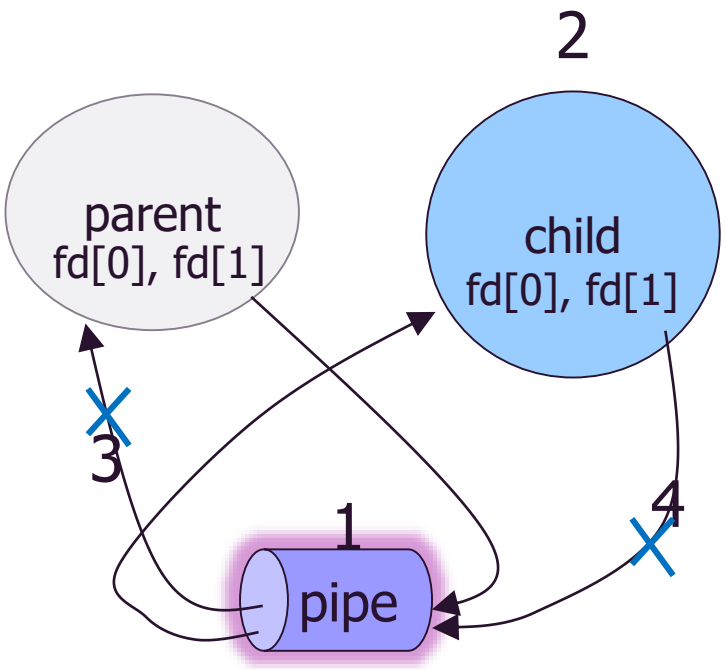
Child sends information to the parent process

Direct Communication Example: Pipe

```c
#include <unistd.h>      // for fork, pipe
i
#include <unistd.h>      // for fork, pipe

int main( void ) {
   enum {RD, WR}; // pipe fd index RD=0, WR=1
   int n, fd[2];
   pid_t pid;
   char buf[100];

   if( pipe(fd) < 0 ) // 1: pipe created
      perror("pipe error");
   else if ((pid = fork()) < 0) // 2: child forked
      perror("fork error");

   else if (pid == 0) {
      close(fd[WR]);// 4: child's fd[1] closed
      n = read(fd[RD], buf, 100);
      write(STDOUT_FILENO, buf, n);
   }

   else {
      close(fd[RD]); // 3: parent's fd[0] closed
      write(fd[WR], "Hello my child\n", 15);
      wait(NULL);
   }
}
}
```

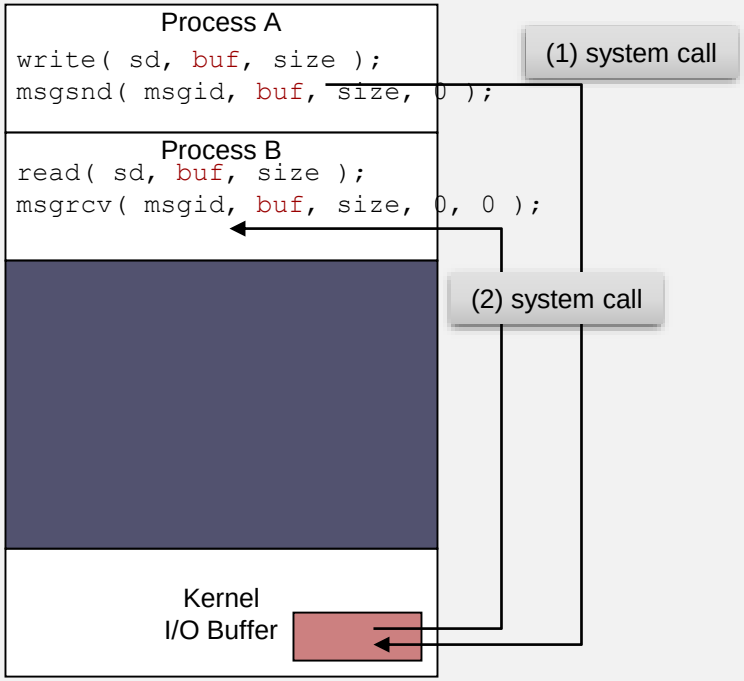
Note: pipe() initializes fd to whatever values are available

Parent sends information to the child process

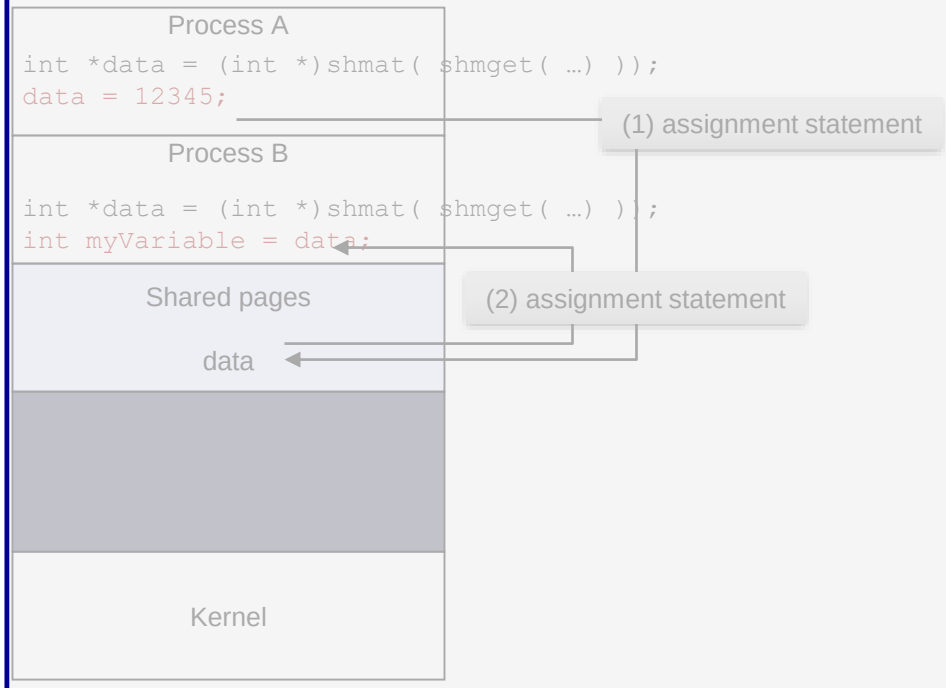


Recall: Communication Models

Message passing



Shared memory



Message Passing: Indirect Communication

U-read

Note: This is out of the scope of our class: this material is covered in Distributed Computing

- 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.
- Processes must know only a mailbox id. They do not need to locate their partners
  - Example: message queue

Inter-Process Synchronization

- Sending Process
  - Blocking – Sender is blocked until message is received or accepted by buffer.
  - Non-Blocking – Sends and resumes execution
- Receiving Process
  - Blocking – Waits until message arrives
  - Non-Blocking – receives valid or NULL

Shared Memory: Buffering

Overview

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link is full (This happens in practical world like sockets).
  3. Unbounded capacity – infinite length
     Sender never waits. (Non-blocking send)

Detail skipped: this is out of the scope of this class, but it is discussed in Parallel/Distributed Computing courses

Shared Memory Example

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    key = 3456;
    shmid = shmget(key, SHMSZ, 0666);
    if (shmid < 0)
    {
        perror("shmget");
        exit(1);
    }
    shm = (char *) shmat(shmid, NULL, 0);

    for (s = shm; *s != 0; s++)
    {
        putchar(*s);
    }
    putchar('\n');

    *shm = '*';
    exit(0);
}
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdio.h>

#define SHMSZ     27
int main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 3456;
    shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
    if (shmid  < 0)
    {
        perror("shmget");
        exit(1);
    }
    shm = (char *) shmat(shmid, NULL, 0);
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
    {
        *s++ = c;
    }
    *s = 0;
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```