"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

Linux Shell

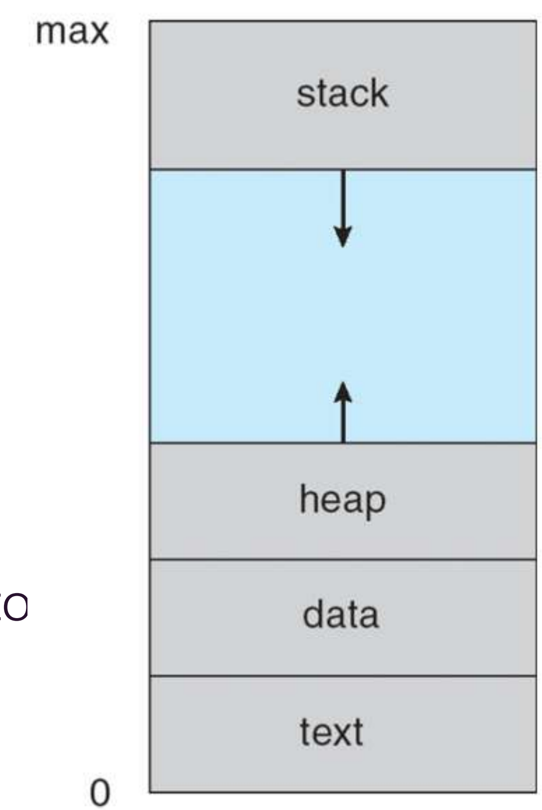
LET'S DISCUSS PROCESSES BEFORE WE STUDY THIS EXAMPLE

Process Management

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

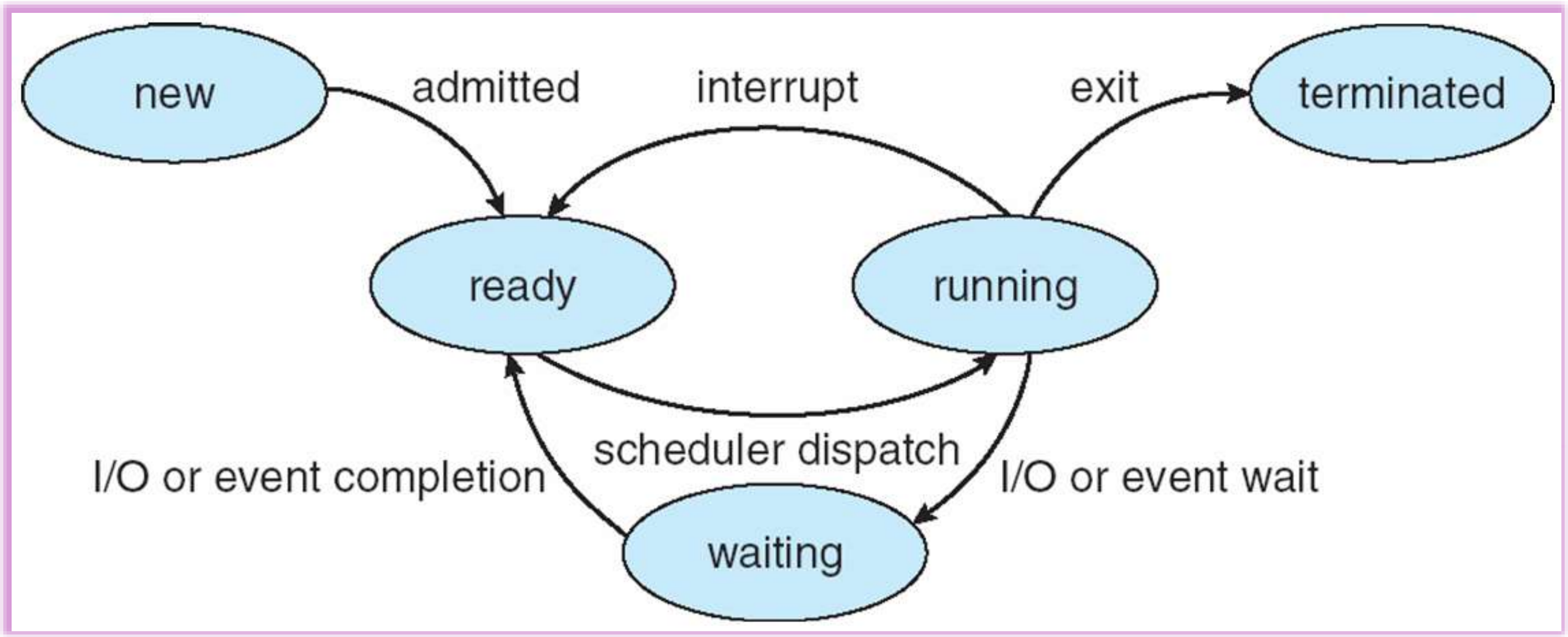
Process Concept

- **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 State

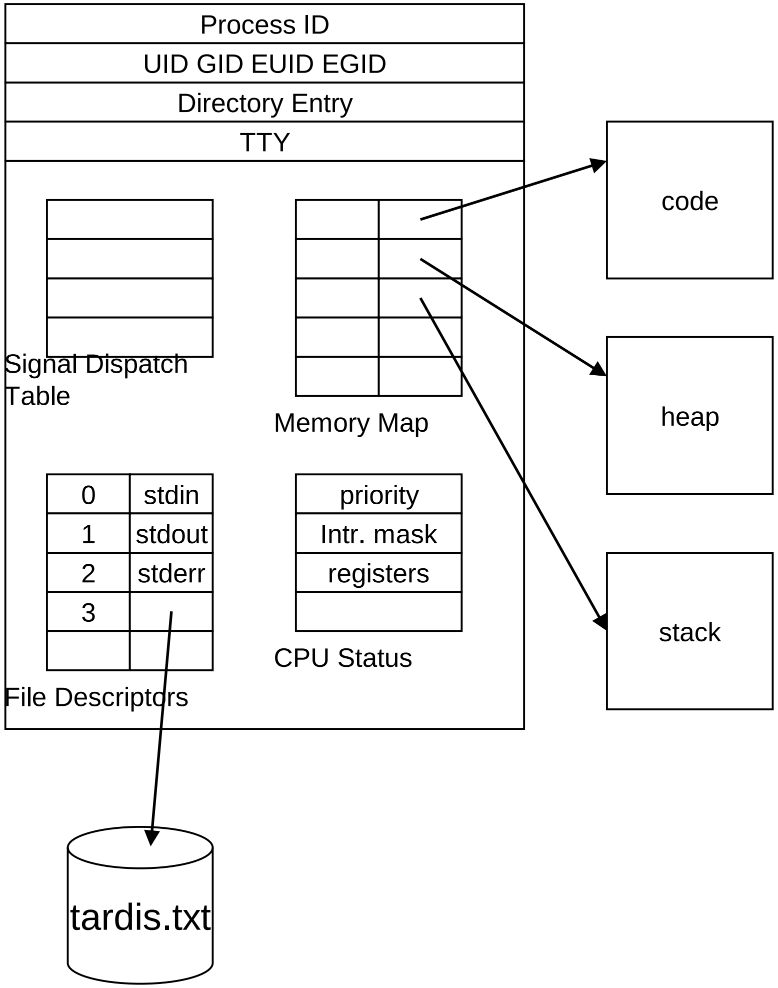
- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process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



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

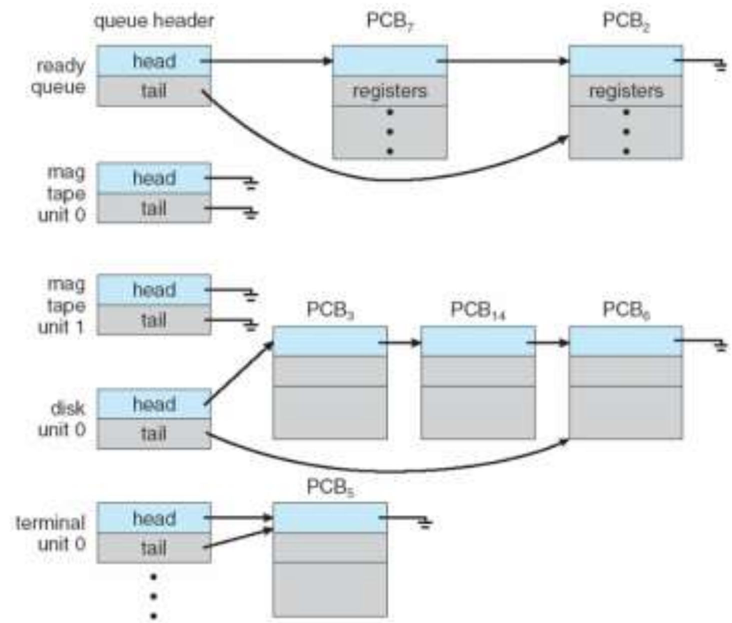


Context Switch: Changing processes

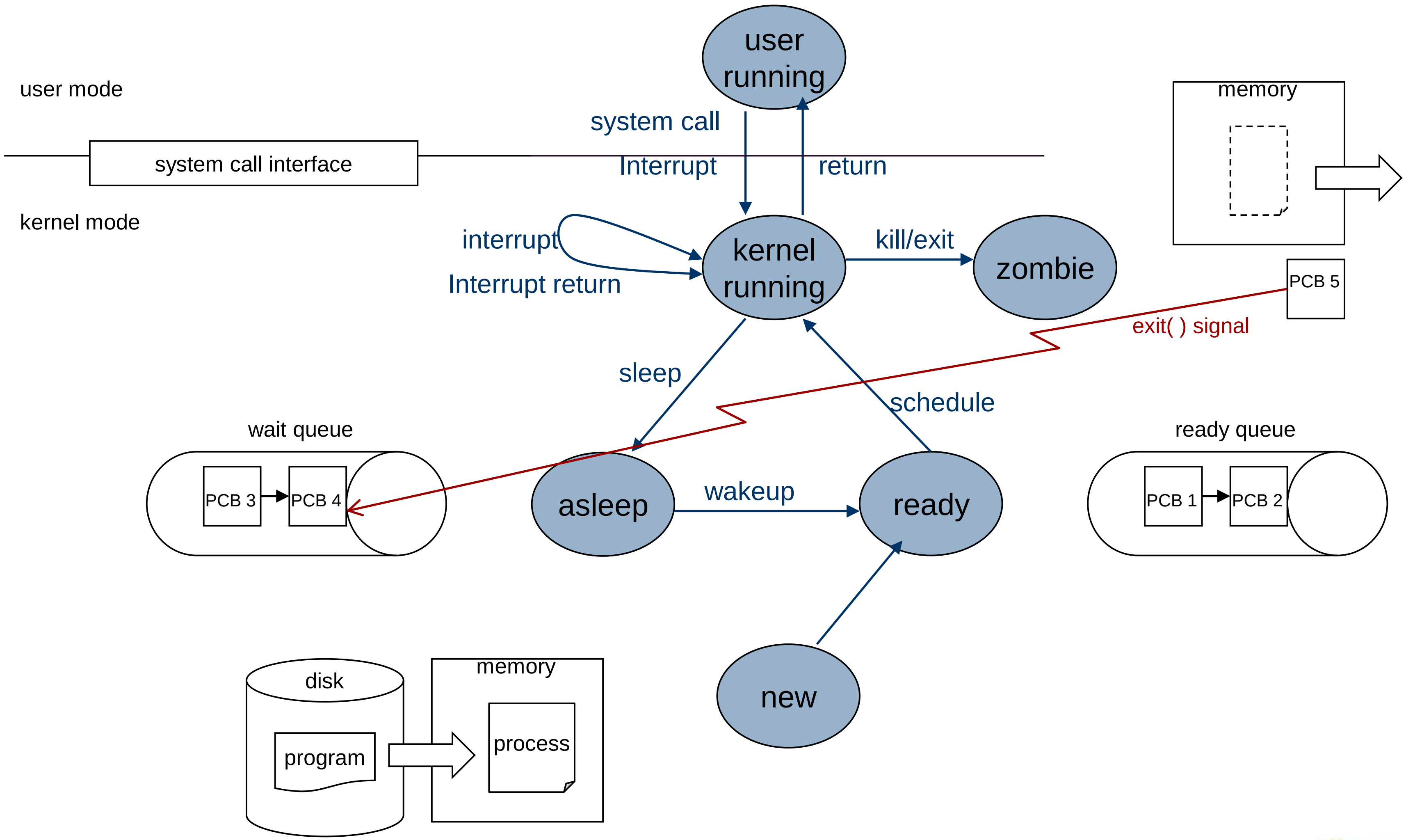
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is **overhead**; the system does no useful work while switching
- Time dependent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

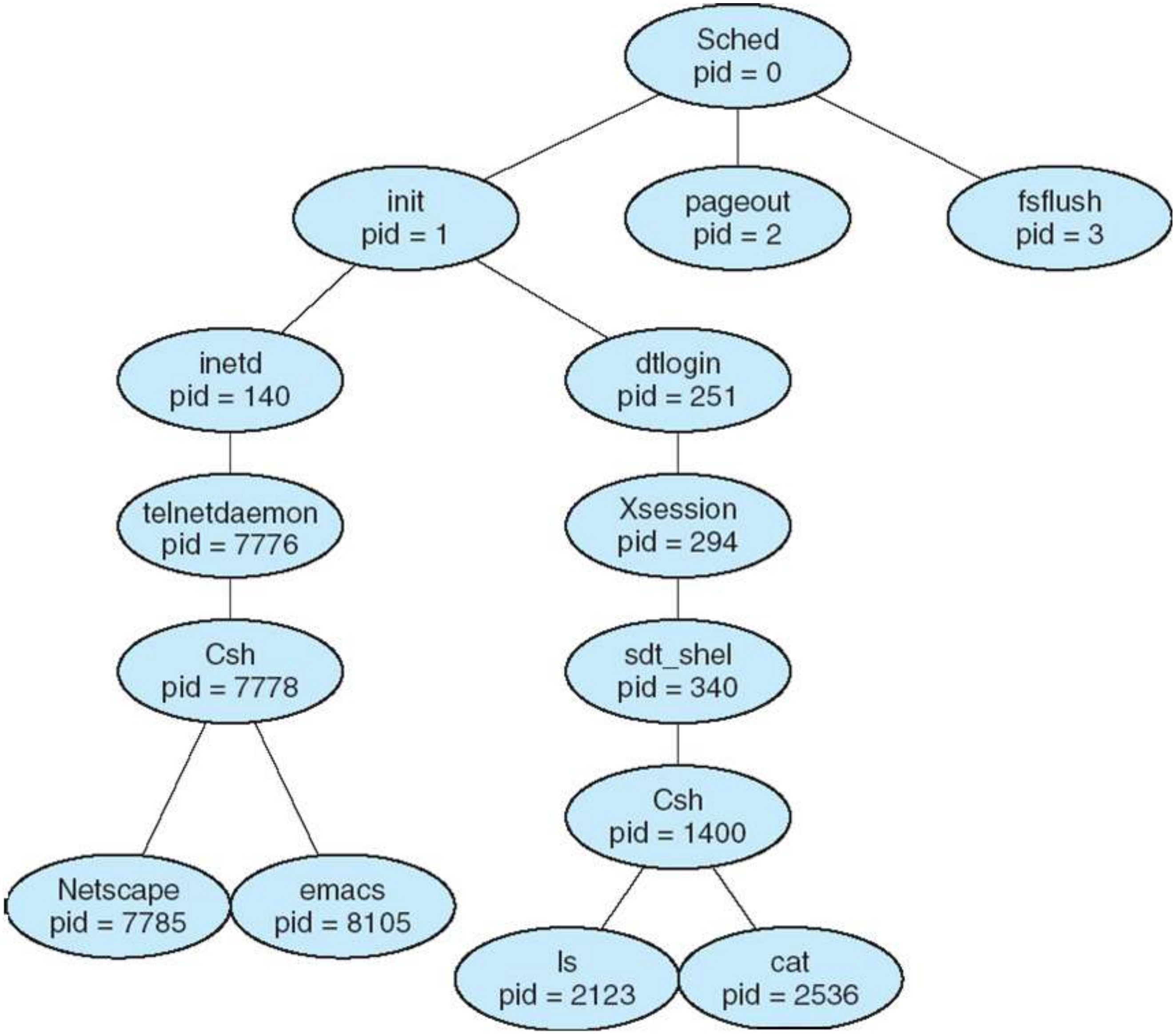
- Thread is a "lightweight process"
- Process consists of CPU state:
  - registers, memory, OS info (open files, PID, etc.),
  - in a thread system there is a larger entity called **task**
- A **task** (or process) consists of memory, OS info and threads
  - Each thread is a unit of execution, consisting 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

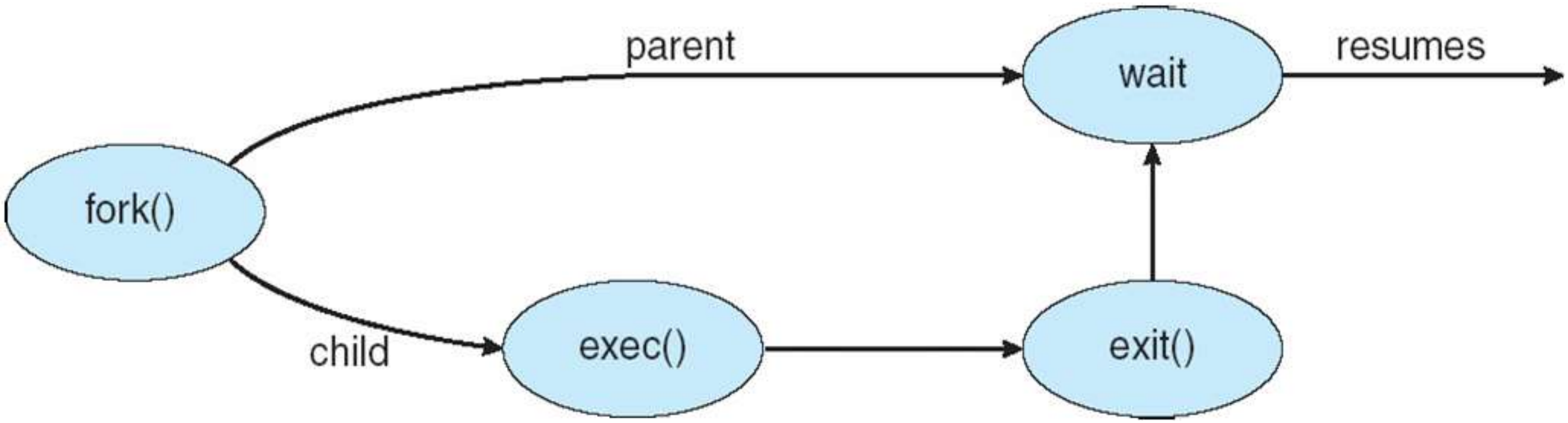
- Parent process creates children processes.
- Resource sharing
  - Resource inherited by children: file descriptors, shared memory and system queues
  - Resource not inherited by children: address space
- Execution
  - Parent and children execute concurrently.
  - Parent waits by **wait** system call until children terminate.
- UNIX 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



C Program Forking Separate 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);
   }
}
```

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.

Discussion 1

- What are the differences between CPU bound and I/O bound processes?
- What is another name for the text section in a process and what does it contain?
- A process can have multiple threads, where would the stack for a thread be allocated from?
- Where does the heap in Java reside?
- What are the lengths of typical operating system scheduling quanta?

Discussion 2

1. List tasks required for process creation and termination in a chronological order.

2. Process 0 in Linux is the ancestor of all processes. Consider tasks performed by Process 0.

3. Upon exit( ), a child process remains as a zombie process that passes an exit code back to its parent. From a Linux shell, you can start a process with & and thus keep it running even after the logoff. What happened to this process?

4. The child process inherits most computing resources from its parent process. Why is it a natural idea? How can this inheritance be implemented inside the operating system?

Cooperating Processes

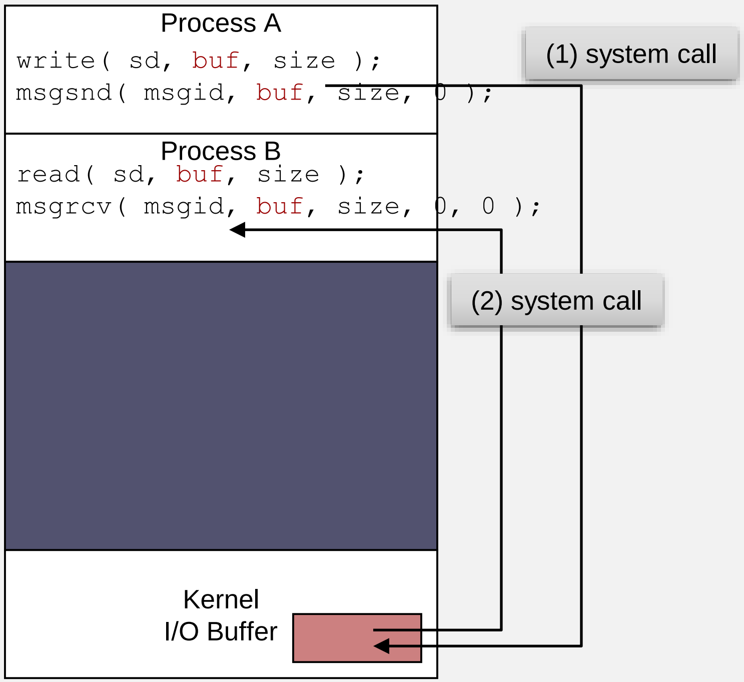
- *Process independence*: Processes belonging to a different user do not affect each other unless they give each other some access permissions

- *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)

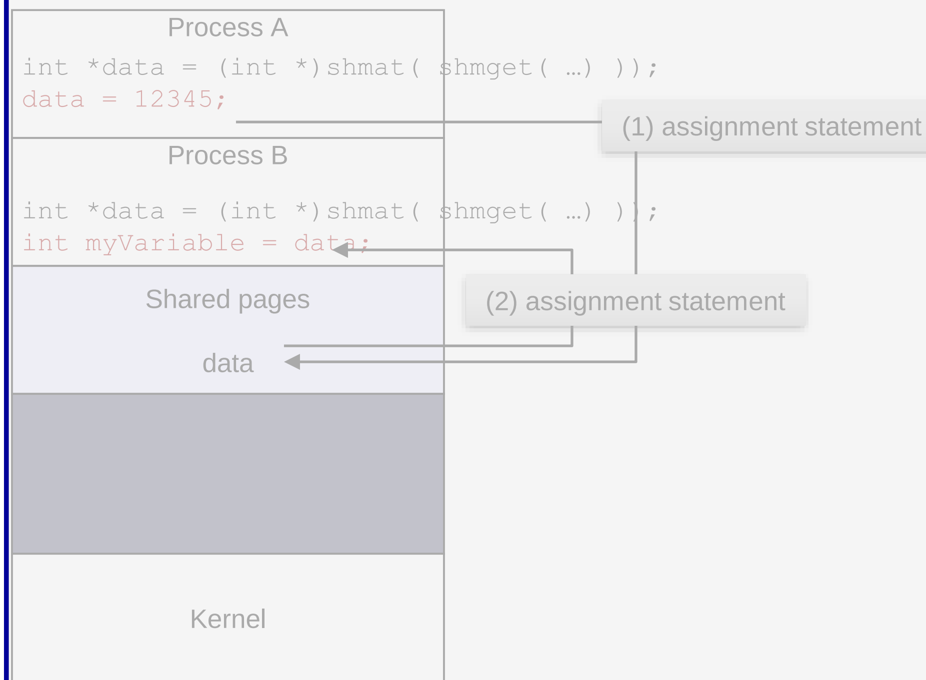
- Advantages of process cooperation
  - Information sharing: (sharing files)
  - Computation speed-up: (parallel programming)
  - Modularity: (like `who | wc -l`, one process lists current users and another counts the number of users.)
  - Convenience: (net-surfing while working on programming with emacs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

Communication Models

- Message passing



- Shared memory



Message Passing

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

- IPC 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

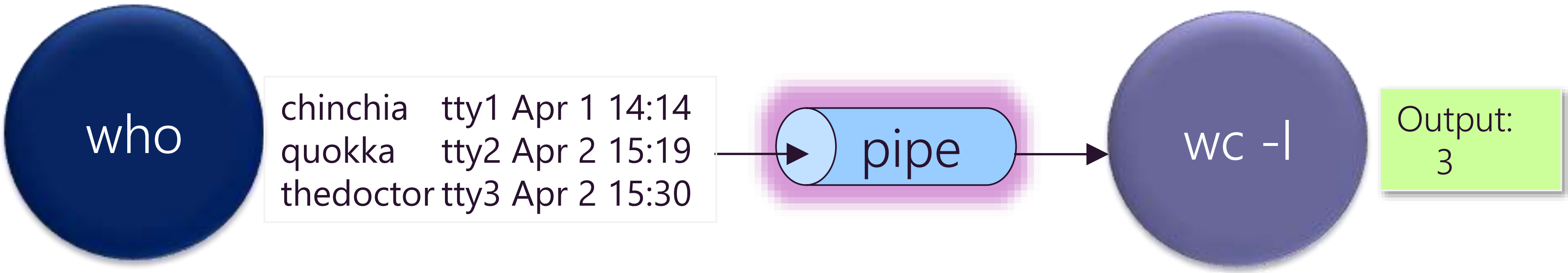
- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q

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

    Example: pipe

Producer-Consumer Problems

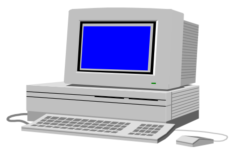
`who | wc -l`



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

Linux Shell

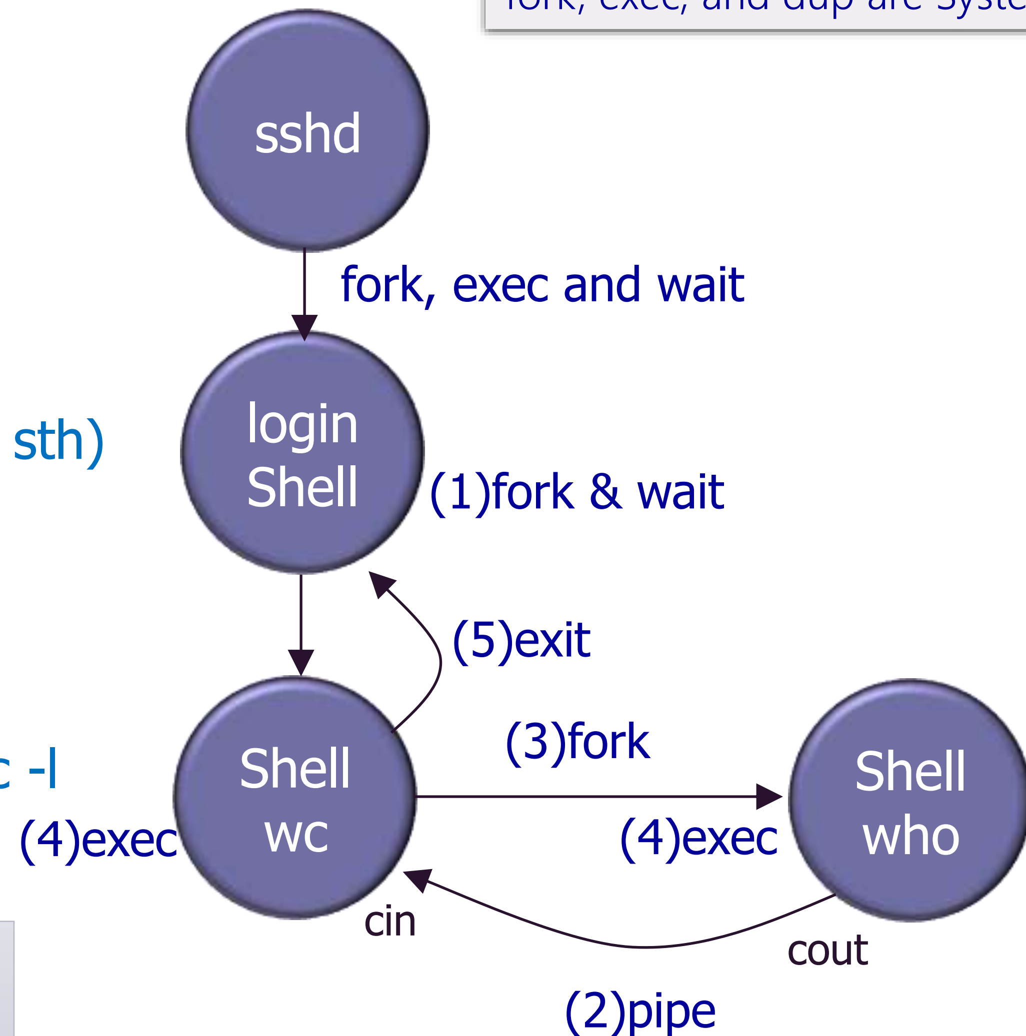
fork, exec, and dup are System calls



mirkwood login: chinchia

mirkwood[1]% (you type sth)

mirkwood[1]% who | wc -l



Child sends information to the parent process

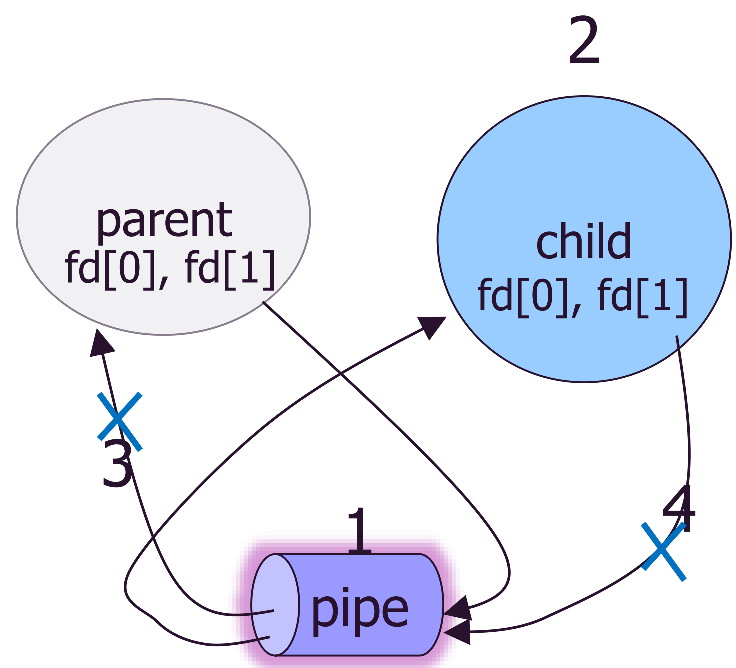
Direct Communication Example: Pipe

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

```c
#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);
   }
}
```

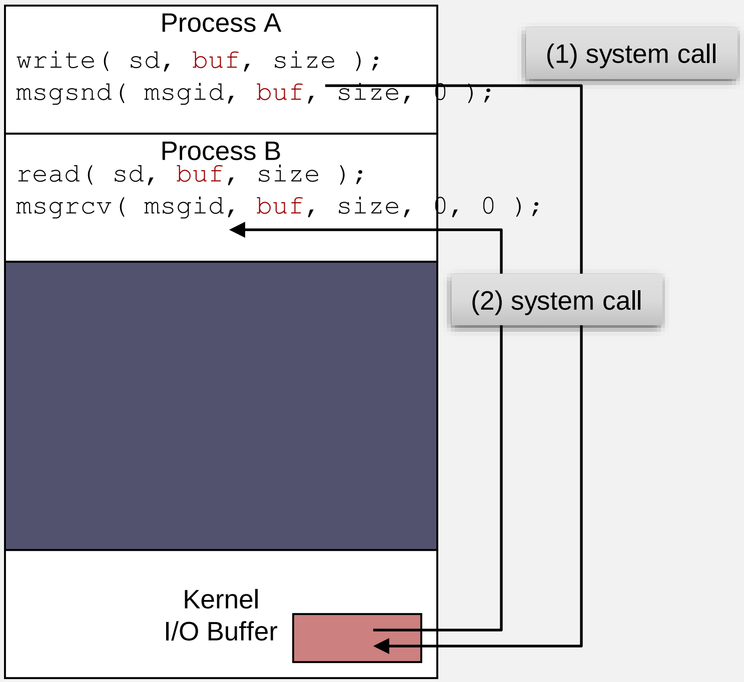
Note: pipe() initializes fd to whatever values are available

Parent sends information to the child process

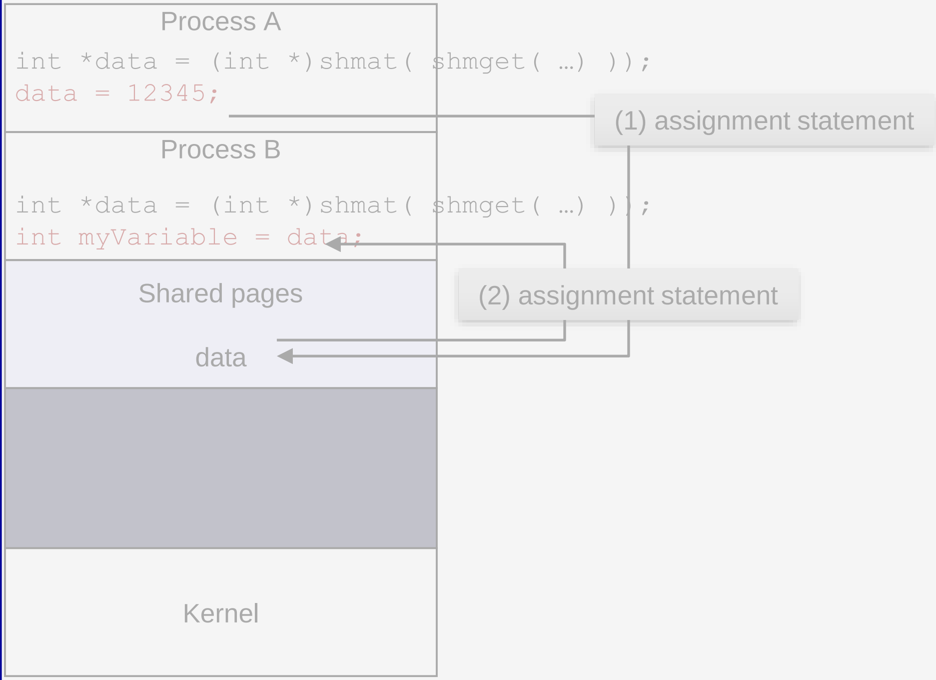


Recall: Communication Models

- Message passing



- Shared memory



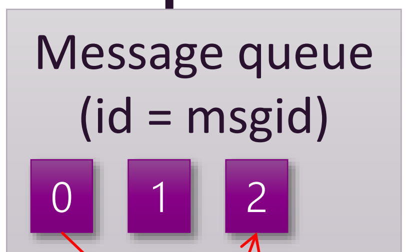
Message Passing: Indirect Communication

- 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

Indirect Communication Example

msg_snd.cpp → msg_rcv.cpp

```cpp
#include <stdlib.h>  // for exit
#include <stdio.h>   // for perror
#include <sys/ipc.h> // for IPC_CREAT
#include <sys/msg.h> // for msgget, msgrcv
#include <iostream>  // for cin, cout, cerr
#include <string.h>  // for strcpy, strlen
using namespace std;
#define MSGSZ 128

typedef struct {
   long mtype;
   char mtext[MSGSZ];
} message_buf;

main() {
   int msqid;
   size_t msgsz;
   int msgflg = IPC_CREAT | 0666;
   message_buf mymsg;

   key_t key = 74563;
   if((msqid = msgget(key, msgflg)) < 0) {
      perror("msgget");
      exit(1);
   }
   cout << "Sending to Msg Queue:" << key << endl;
   strcpy( mymsg.mtext, "Hello Huskies!");
   msgsz = strlen(mymsg.mtext) + 1;
   mymsg.mtype = 1;

   if(msgsnd(msqid,&mymsg,msgsz,IPC_NOWAIT) < 0)
   {
      perror("msgsnd");
   }
}
```



Some other process may enqueue and dequeue a message

```cpp
#include <stdlib.h>  // for exit
#include <stdio.h>   // for perror
#include <sys/ipc.h> // for IPC_CREAT
#include <sys/msg.h> // for msgget, msgrcv
#include <iostream>  // for cin, cout, cerr
using namespace std;
#define MSGSZ 128

typedef struct {
   long mtype;
   char mtext[MSGSZ];
} message_buf;

main() {
   int msqid;
   size_t msgsz;
   int msgflg = IPC_CREAT | 0666;
   message_buf mymsg;

   key_t key = 74563;
   if((msqid = msgget(key, msgflg)) < 0) {
      perror("msgget");
      exit(1);
   }

   if(msgrcv(msqid, &mymsg, MSGSZ, 1, 0) < 0) {
      perror("msgrcv");
      exit(1);
   }
   cout << mymsg.mtext << endl;
}
```

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: Discussed in CSS434: Parallel/Distributed Comp.

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);
}
```

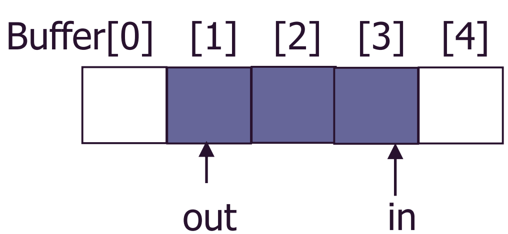
Producer and Consumer Example

Producer Process

```
for(int i = 0; ; i++ )
{
   BoundedBuffer.enter(new Integer(i));
}
```

```
public void enter( Object item )
{
   while ( count == BUFFER_SIZE )
     ; // buffer is full! Wait till buffer is consumed
   ++count;
   buffer[in] = item; // add an item
   in = ( in + 1 ) % BUFFER_SIZE;
}
```

```
public object remove( )
{
   Object item;
   while ( count == 0 )
     ; // buffer is empty! Wait till buffer is filled
   --count;
   item = buffer[out]; // pick up an item
   out = ( out + 1 ) % BUFFER_SIZE;
}
```



Consumer Process

```
for(int i = 0; ; i++ )
{
   BoundedBuffer.remove( );
}
```

Discussion

1. In Unix, the first process is called **init**. All the others are descendants of "init". The **init** process spawns a **sshd** process that detects a new ssh connection. Upon a new connection, **sshd** spawns a login process that then overloads a **shell** on it when a user successfully log in the system. Now, assume that the user types **who | grep chinchia | wc –l**. Draw a process tree from **init** to those three commands. Add **fork**, **exec**, **wait**, and **pipe** system calls between any two processes affecting each other.

2. Consider four different types of inter-process communication.
   a) Pipe: implemented with pipe, read, and write
   b) Socket: implemented with socket, read, and write
   c) Shared memory: implemented shmget, shmat, and memory read/write
   d) Shared message queue: implemented with msgget, msgsnd, and msgrcv

   1. Which types are based on direct communication?
   2. Which types of communication do not require parent/child process relationship?
   3. If we code a produce/consumer program, which types of communication require us to implement process synchronization?
   4. Which types of communication can be used to communicate with a process running on a remote computers?
   5. Which types of communication must use file descriptors?
   6. Which types of communication need a specific data structure when transferring data?