CSCI 6730/ 4730
Operating Systems

Processes

Review

- Operating System Fundamentals
  - » What is an OS?
  - » What does it do?
  - » How and when is it invoked?
- Structures
  - » Monolithic
  - » Layered
  - » Microkernels
  - » Virtual Machines
  - » Modular

Chapter 3: Processes: Outline

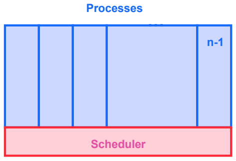
- Process Concept: views of a process
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

What is a Process?

- A process is a program in execution (an active entity, i.e. it is a *running* program)
  - » Basic unit of work on a computer, a job, a task.
  - » A container of instructions with some resources:
    - – e.g. CPU time (CPU carries out the instructions), memory, files, I/O devices to accomplish its task
  - » Examples: compilation process, word processing process, scheduler (`sched`, `swapper`) process or daemon processes: `ftpd`, `httpd`
- System view…

What are Processes?

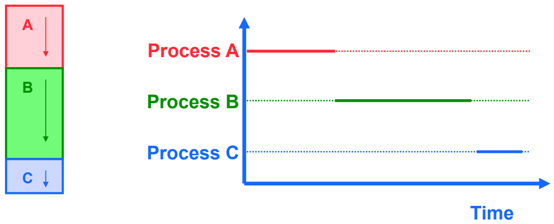
- Multiple processes:
  - » Several distinct processes can execute the SAME program
- Time sharing systems run several processes by multiplexing between them
- ALL “runnables” including the OS are organized into a number of “sequential processes”



Our Process Definition

A process is a ‘program in execution’, a sequential execution characterized by trace. It has a context (the information or data) and this ‘context’ is maintained as the process progresses through the system.

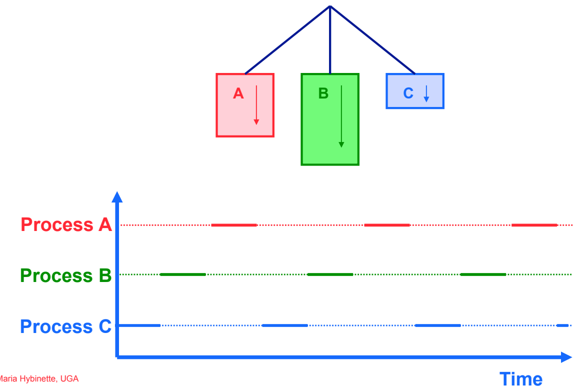
Activity of a Process



Multiprogramming:

- Solution: provide a programming counter.
- One processor (CPU).

Activity of a Process: Time Sharing

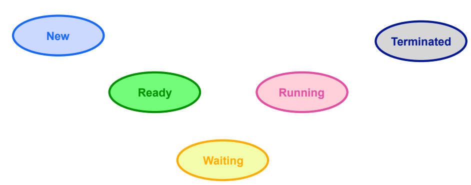


What Does the Process Do?

- Created
- Runs
- Does not run (but ready to run)
- Runs
- Does not run (but ready to run)
- ….
- Terminates

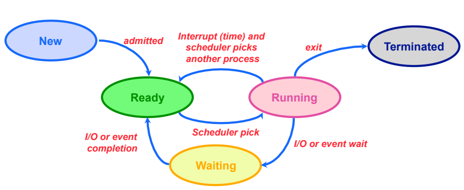
'States' of a Process

- As a process executes, it changes *state*
  - » New: The process is being created.
  - » Running: Instructions are being executed.
  - » Ready: The process is waiting to be assigned to a processor (CPU).
  - » Terminated: The process has finished execution.
  - » Waiting: The process is waiting for some event to occur.



State Transitions

- A process may change state as a result:
  - » Program action (system call)
  - » OS action (scheduling decision)
  - » External action (interrupts)



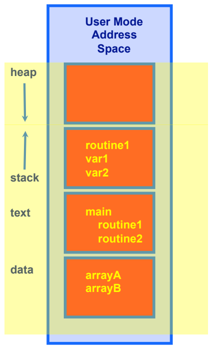
OS Designer's Questions?

- How is process state represented?
  - » What information is needed to represent a process?
- How are processes selected to transition between states?
- What mechanism is needed for a process to run on the CPU?

What Makes up a Process?

User resources/OS Resources:

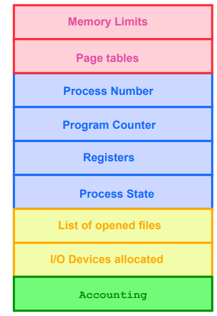
- Program code (text)
- Data
  - » global variables
  - » heap (dynamically allocated memory)
- Process stack
  - » function parameters
  - » return addresses
  - » local variables and functions
- OS Resources, environment
  - » open files, sockets
  - » Credential for security
- Registers
  - » program counter, stack pointer



address space are the shared resources of a(ll) thread(s) in a program

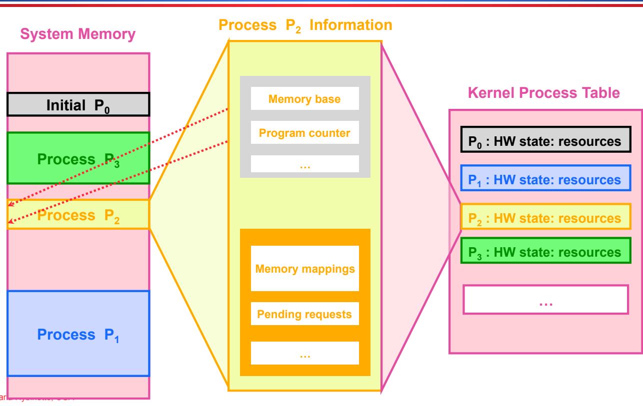
What is needed to keep track of a Process?

- Memory information:
  - » Pointer to memory segments needed to run a process, i.e., pointers to the address space -- text, data, stack segments.
- Process management information:
  - » Process state, ID
  - » Content of registers:
    - – Program counter, stack pointer, process state, priority, process ID, CPU time used
- File management & I/O information:
  - » Working directory, file descriptors open, I/O devices allocated
- Accounting: amount of CPU used.



Process control Block (PCB)

Process Representation



OS View: Process Control Block (PCB)

- How does an OS keep track of the state of a process?
  - » Keep track of 'some information' in a structure.
    - – Example: In Linux a process' information is kept in a structure called `struct task_struct` declared in `#include/linux/sched.h`
    - – What is in the structure?

```
struct task_struct
    pid_t pid;                 /* process identifier */
    long state;                /* state for the process */
    unsigned int time_slice    /* scheduling information */
    struct mm_struct *mm       /* address space of this process */
```


State in Linux

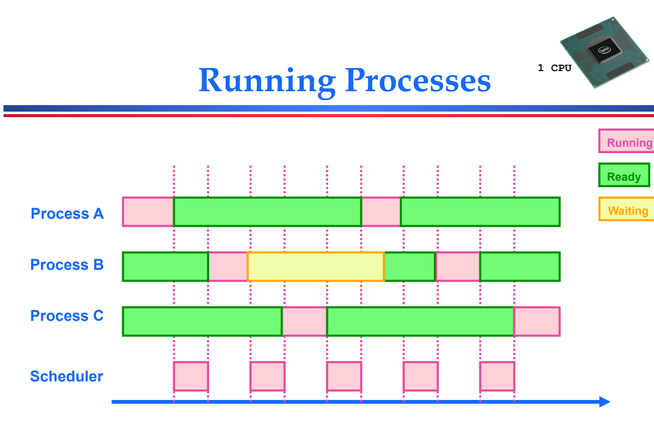
```
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_EXCLUSIVE          32
```

• traditionally 'zombies' are child processes of parents that have not processed a wait() instruction.

• Note: processes that have been 'adopted' by init are not zombies (these are children of parents that terminates before the child). Init automatically calls wait() on these children when they terminate.

• this is true in LINUX.
•What to do: 1) Kill the parent 2) Fix the parent (make it issue a wait) 2) Don't care

Running Processes



Why is Scheduling important?

- **Goals:**
  - » **Maximize the 'usage' of the computer system**
  - » **Maximize CPU usage (utilization)**
  - » **Maximize I/O device usage**
  - » **Meet as many task deadlines as possible (maximize throughput).**

Scheduling

- **Approach: Divide up scheduling into task levels:**
  - » **Select process who gets the CPU (from main memory).**
  - » **Admit processes into memory**
    - – **Sub problem: How?**
- **Short-term scheduler (CPU scheduler):**
  - » **selects which process should be executed next and allocates CPU.**
  - » **invoked frequently (ms) ⇒ (must be fast).**
- **Long-term scheduler (look at first):**
  - » **selects which processes should be brought into the memory (and into the ready state)**
  - » **invoked infrequently (seconds, minutes)**
  - » **controls the *degree of multiprogramming.***

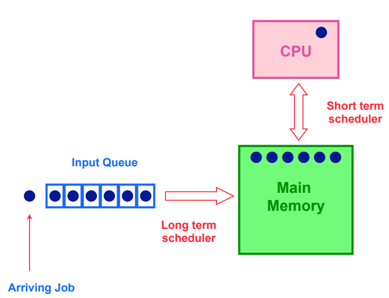
Process Characteristics

- **Processes can be described as either:**
  - » **I/O-bound process – spends more time doing I/O than computations, many short CPU bursts.**
  - » **CPU-bound process – spends more time doing computations; few very long CPU bursts.**

Observations

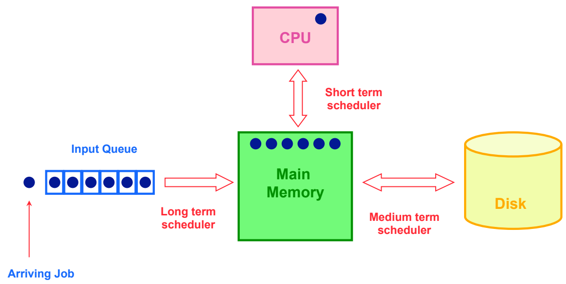
- **If all processes are I/O bound, the ready queue will almost always be empty (little scheduling)**
- **If all processes are CPU bound the I/O devices are underutilized**
- **Approach (long term scheduler): 'Admit' a good mix of CPU bound and I/O bound processes.**

Big Picture (so far)



Exhaust Memory?

- **Problem: What happens when the number of processes is so large that there is not enough room for all of them in memory?**
- **Solution: Medium-level scheduler:**
  - » **Introduce another level of scheduling that removes processes from memory; at some later time, the process can be reintroduced into memory and its execution can be continued where it left off**
  - » **Also affect degree of multi-programming.**

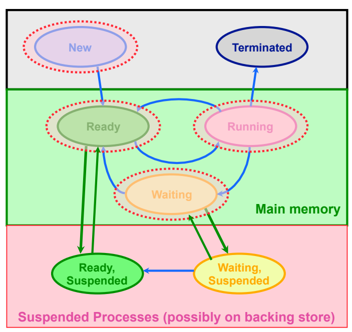


Which processes should be selected?

- Processor (CPU) is faster than I/O so all processes could be waiting for I/O
  - » Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
  - » Two new states
    - – waiting, suspend
    - – Ready, suspend

Suspending a Process

- Which to suspend?
- Others?



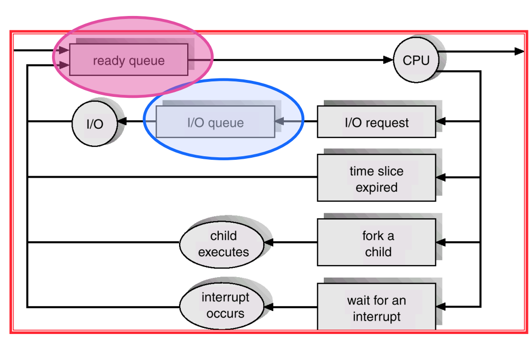
Possible Scheduling Criteria

- How long since process was swapped in our out?
- How much CPU time has the process had recently?
- How big is the process (small ones do not get in the way)?
- How important is the process (high priority)?

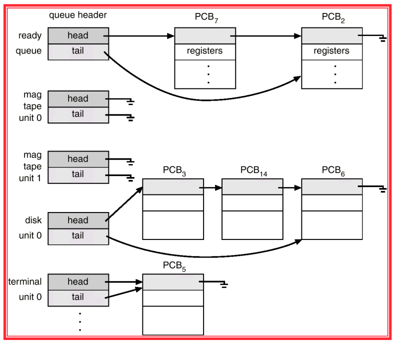
OS Implementation: Process Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute on CPU
- **Device queues** – set of processes waiting for an I/O device.
- Process migration between the various queues.

Representation of Process Scheduling



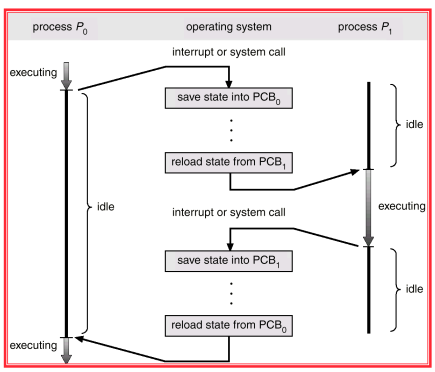
Ready Queue, I/O Device Queues



Context Switch

- 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.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

CPU Context Switches



Process Creation

- **Process Cycle**: Parents create children; results in a tree of processes.
- Address space models:
  - » Child duplicate of parent.
  - » Child has a program loaded into it.
- Execution models:
  - » Parent and children execute concurrently.
  - » Parent waits until children terminate.
- Examples

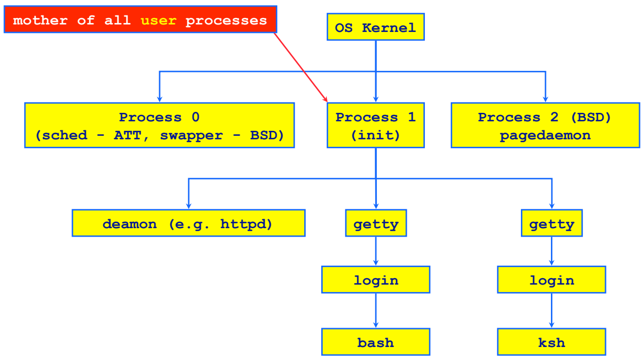
Continuing the Boot Sequence…

- After loading in the Kernel and it does a number of system checks it creates a number of 'dummy processes' -- processes that cannot be killed -- to handle system tasks.
- Usually ….

Process Life Cycle: UNIX (cont)

- PID 0 is *usually* the `scheduler` process (often called `swapper`)
  - » is a system process -- it is part of the kernel
  - » the grandmother of all processes).
- `init` - Mother of all user processes, init is started at boot time (at end of the boot strap procedure) and is responsible for starting other processes
  - » It is a user process (not a system process that runs within the kernel like `swapper`) with PID 1 (but runs with root privileges)
  - » `init` uses file `inittab` and directory `/etc/rc?.d`
  - » brings the user to a certain specified state (e.g., multiuser mode)
- `getty` - login process that manages login sessions

Processes Tree on a UNIX System



Other Systems

```
HP-UX 10.20

UID     PID  PPID C  STIME   TTY     TIME COMMAND
  root    0     0  0  Apr 20  ?       0:17 swapper
  root    1     0  0  Apr 20  ?       0:00 init
  root    2     0  0  Apr 20  ?       1:02 vhand
```

Page handler
Process spawner
Scheduler
Buffering/Flushing I/O

```
Linux RedHat 6.0:

  UID      PID  PPID  C STIME TTY      TIME CMD
  root       1     0  0 09:59 ?     00:00:07 init
  root       2     1  0 09:59 ?     00:00:00 [kflushd]
  root       3     1  0 09:59 ?     00:00:00 [kpiod]
  root       4     1  0 09:59 ?     00:00:00 [kswapd]
  root       5     1  0 10:00 ?     00:00:00 [mdrecoveryd]
```

```
Solaris:

  UID   PID  PPID  C   STIME TTY     TIME CMD
  root    0     0  0   Apr 19 ?      0:00 sched
  root    1     0  0   Apr 19 ?      0:22 /etc/init -
  root    2     0  0   Apr 19 ?      0:00 pageout
```

* sched – dummy process which provides swapping services
* pageout – dummy process which provides virtual memory (paging) services

Running Processes

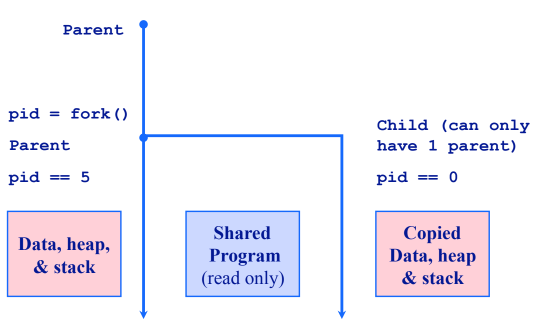
- Print out status information of various processes in the system: `ps -axj` (BSD) , `ps -efjc` (SVR4)
- Daemons (background processes) with root privileges, no controlling terminal, parent process is `init`

```
{atlas:maria} ps -efjc | sort -k 2 -n | more
    UID   PID PPID  PGID SID  CLS PRI    STIME TTY   TIME CMD
   root     0    0     0   0  SYS  96   Mar 03 ?     0:01 sched
   root     1    0     0   0   TS  59   Mar 03 ?     1:13 /etc/init -r
   root     2    0     0   0  SYS  98   Mar 03 ?     0:00 pageout
   root     3    0     0   0  SYS  60   Mar 03 ? 4786:00 fsflush
   root    61    1    61  61   TS  59   Mar 03 ?     0:00 /usr/lib/sysevent/syseventd
   root    64    1    64  64   TS  59   Mar 03 ?     0:08 devfsadmd
   root    73    1    73  73   TS  59   Mar 03 ?   30:29 /usr/lib/picl/picld
   root   256    1   256 256   TS  59   Mar 03 ?     2:56 /usr/sbin/rpcbind
   root   259    1   259 259   TS  59   Mar 03 ?     2:05 /usr/sbin/keyserv
   root   284    1   284 284   TS  59   Mar 03 ?     0:38 /usr/sbin/inetd -s
 daemon   300    1   300 300   TS  59   Mar 03 ?     0:02 /usr/lib/nfs/statd
   root   302    1   302 302   TS  59   Mar 03 ?     0:05 /usr/lib/nfs/lockd
   root   308    1   308 308   TS  59   Mar 03 ?  377:42 /usr/lib/autofs/automountd
   root   319    1   319 319   TS  59   Mar 03 ?     6:33 /usr/sbin/syslogd
```


Process Creation: Execution & Address Space in UNIX

- In UNIX process `fork()-exec()` mechanisms handles process creation and its behavior:
  - » `fork()` creates an exact copy of itself (the parent) and the new process is called the child process
  - » `exec()` system call places the image of a new program over the newly copied program of the parent

`fork()` a child



Example: `parent-child.c`

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

int main()
  {
   int i;
   pid_t pid;
   pid = fork();
   if( pid > 0 )
     {                 /* parent */
     for( i = 0; i < 1000; i++ )
            printf( "\tPARENT %d\n", i );
         }
   else
         {             /* child */
         for( i = 0; i < 1000; i++ )
              printf( "\t\tCHILD %d\n", i );
         }
   }
```

```
{saffron} parent-child
          PARENT 0
          PARENT 1
          PARENT 2
                   CHILD 0
                   CHILD 1
          PARENT 3
          PARENT 4
                   CHILD 2
          .
          .
```

Things to Note

- `i` is copied between parent and child
- The switching between parent and child depends on many factors:
  - » Machine load, system process scheduling, …
- I/O buffering effects the output shown
  - » Output interleaving is *non-deterministic*
    - – Cannot determine output by looking at code

Process Creation: Windows

- Processes created via 10 params `CreateProcess()`
- Child process *requires* loading a specific program into the address space.

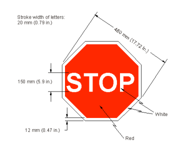
```
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,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


Process Termination

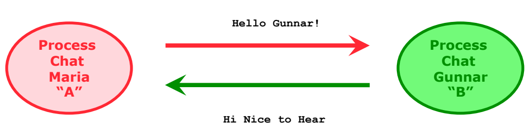
- Process executes last statement and asks the operating system to delete it by using the `exit()` system call.
  - » Output data from child to parent (via wait).
  - » Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (abort).
  - » Child has exceeded allocated resources.
  - » Task assigned to child is no longer required.
  - » Parent is exiting.
    - – Some Operating system does not allow child to continue if its parent terminates.
      - Cascading termination (initiated by system to kill of children of parents that exited).
    - – If a parents terminates children are adopted by init() - so they still have a parent to collect their status and statistics

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
  - » Advantages of process cooperation
    - – Information sharing
    - – Computation speed-up
    - – Modularity
    - – Convenience
  - » Requirement: Inter-process communication (IPC) mechanism.



Two Communicating Processes



- Concept that we want to implement

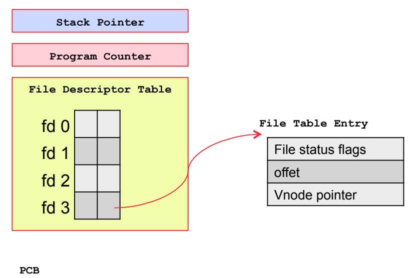
On the path to communication…

- Want: A communicating processes
- Have so far: Forking – to create processes
- Problem:
  - » After fork() is called we end up with two independent processes.
  - » Separate Address Spaces
- Solution? How do we communicate?

File: The Unix Way

- One easy way to communicate is to use files.
  - » Process A writes to a file and process B reads from it
- File descriptors
  - » Mechanism to work with files
  - » Used by low level I/O
    - – Open(), close(), read(), write()
  - » file descriptors generalize to other communication devices such as pipes and sockets

Big Picture



Producer Consumer Problems

- Simple example: who | sort
  - » Both the writing process (who) and the reading process (sort) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS *buffer* with 2 file descriptors.
  - » It is a resource that is concurrently accessed
    - – by the reader and the writer, so it must be managed carefully (by the Kernel)

Producer / Consumer: Buffering

- Un-buffered – output appears immediately stderr is not buffered
- Line buffered – output appears when a full line has been written.
  - » stdout is line buffered when going to the screen
- Block buffered – output appears when a buffer is filled or a buffer is flushed (on close or explicit flush).
  - » normally output to a file is block buffered
  - » stdout is block buffered when redirected to a file.

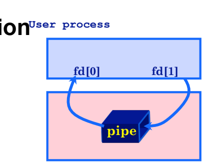
Producer / Consumer: Buffering

- Consumer blocks when buffer is empty
- Producer blocks when buffer is full
- Producer and Consumer should run independently as far as buffer capacity and contents permit
- They should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- Harder problem if there is more than one consumer and/or more than one producer

Buffering: Programming with Pipes

```
#include <unistd.h>
int pipe( int fd[2] );
```

- `pipe()` binds `fd[]` with two file descriptors:
  - » `fds[0]` used to read from pipe
  - » `fds[1]` used to write to pipe
- Half-Duplex (one way) Communication
- Returns 0 if OK and -1 on error.

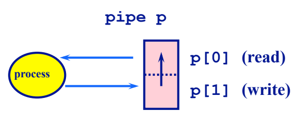


Example: pipe-yourself.c

```c
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE  16   /* null */

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

int main()
   {
   char inbuf[MSGSIZE];
   int  p[2], i;

   if( pipe( p ) < 0 )
     {  /* open pipe */
        perror( "pipe" );
        exit( 1 );
     }
```

```c
write( p[1], msg1, MSGSIZE );
write( p[1], msg2, MSGSIZE );
write( p[1], msg3, MSGSIZE );

for( i=0; i < 3; i++ )
   {  /* read pipe */
   read( p[0], inbuf, MSGSIZE );
   printf( "%s\n", inbuf );
   }
return 0;
}
```

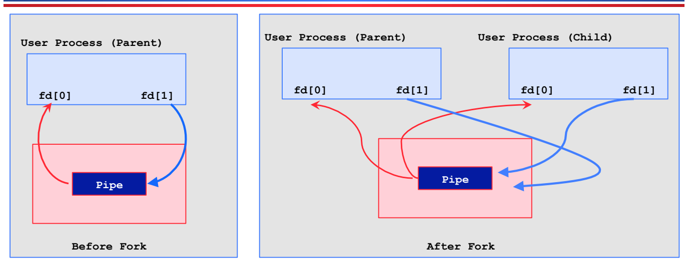
```
{saffron:ingrid:4} pipe-yourself
hello, world #1
hello, world #2
hello, world #3
```



Things to Note

- Pipes uses FIFO ordering: *first-in first-out*.
- Read / write amounts do not need to be the same, but then text will be split differently.
- Pipes are most useful with `fork()` which creates an IPC connection between the parent and the child (or between the parents children)

What Happens After Fork?



- Decide on : Direction of Data Flow – then close appropriate ends of pipe (at both parent and child)

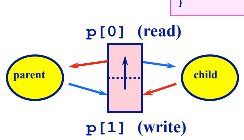
- A forked child inherits file descriptors from its parent
- pipe() creates an internal system buffer and two file descriptors, one for reading and one for writing.
- After the pipe call, the parent and child should close the file descriptors for the opposite direction. Leaving them open does not permit full-duplex communication.

Example: pipe-fork-close.c

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#define MSGSIZE  16

char *msg1="hello, world #1";
char *msg2="hello, world #2";
char *msg3="hello, world #3";

int main()
   {
   char inbuf[MSGSIZE];
   int p[2], i, pid;

   if( pipe( p ) < 0 )
     {  /* open pipe */
        perror( "pipe" );
        exit( 1 );
     }
   if( (pid = fork()) < 0 )
     {
        perror( "fork" );
        exit( 2 );
     }
```

```c
if( pid > 0 ) /* parent */
   {
   close( p[0] ); /* read link */
   write( p[1], msg1, MSGSIZE );
   write( p[1], msg2, MSGSIZE );
   write( p[1], msg3, MSGSIZE );
   wait( (int *) 0 );
   }
```

```c
if( pid == 0 ) /* child */
   {
   close( p[1] ); /* write link */
   for( i=0; i < 3; i++ )
      {
      read( p[0], inbuf, MSGSIZE );
      printf( "%s\n", inbuf );
      }
   }
return 0;
}
```



Some Rules of Pipes

- Every pipe has a size limit
  - » `POSIX` minimum is 512 bytes -- most systems makes this figure larger
- `read()` blocks if pipe is empty ***and*** there is a a `write` link open to that pipe
- `read()` from a pipe whose `write()` end is closed ***and*** is empty returns 0 (indicates `EOF`)
  - » Close write links or `read()` will never return
- `write()` to a pipe with no `read()` ends returns -1 and generates `SIGPIPE` and `errno` is set to `EPIPE`
- `write()` blocks if the pipe is full or there is not enough room to support the `write()`.
  - » May block in the middle of a `write()`

Pipes and `exec()`

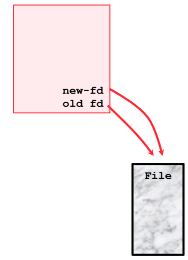
How can we code `who | sort` ?

1. Use `exec()` to start two processes (one runs `who` the other `sort`) which share a pipe.
2. Connect the pipe to `stdin` and `stdout` using `dup2()`.

Duplicate File Descriptors

```
#include <unistd.h>
int dup2( int old-fd, int new-fd );
```

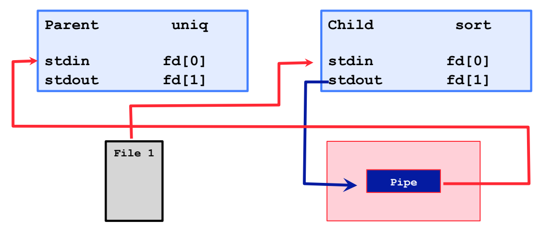
- Set one FD to the value of another.
- new-fd and old-fd now refer to the same file
- if new-fd is open, it is first automatically closed
- Note that dup2() refer to fds not streams



Example : "`sort < file1 | uniq`"

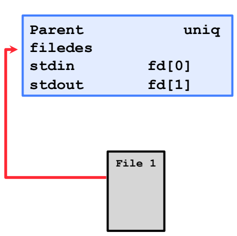
- What does this look like?
  - » Parent
  - » Child

Want: "`sort < file1 | uniq`"



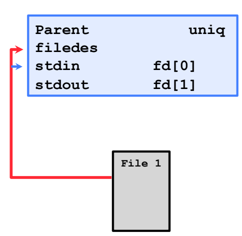
- Want: How do we get there?

Want: "`sort < file1 | uniq`"



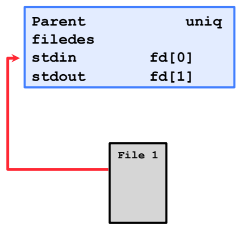

```
fileDES = open( "myfile.txt", O_RDONLY );
```


Want: "`sort < file1 | uniq`"



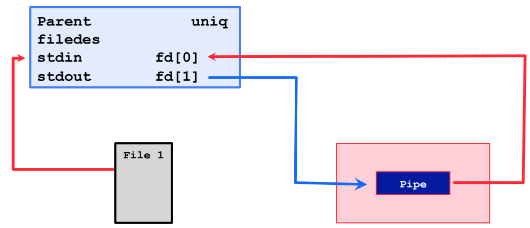

```
fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );
```

Want: “sort < file1 | uniq”



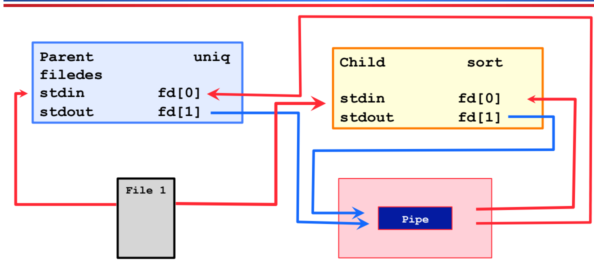

```
fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );
close( fileDES );
```


Want: “sort < file1 | uniq”



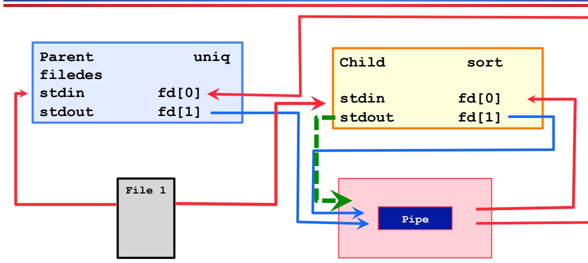

```
pipe( fd );
```


Want: “sort < file1 | uniq”



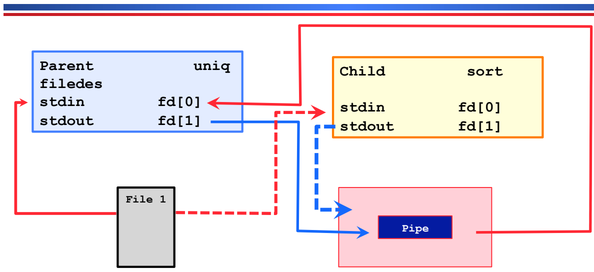

```
fork();
```


Want: “sort < file1 | uniq”



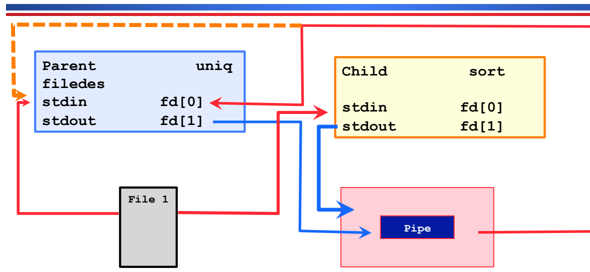

```
dup2(fd[1], fileno(stdout)); /* in green */
```


Want: “sort < file1 | uniq”



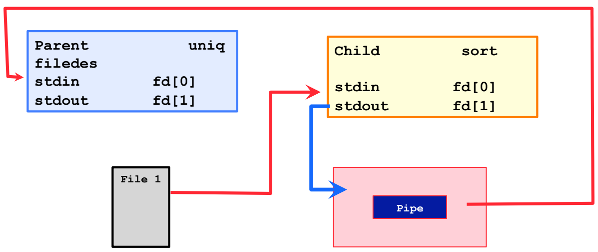

```
close(fd[0]); close(fd[1]);  /* child */
/* leaving the ---- connections for child */
```


Want: “sort < file1 | uniq”




```
dup2(fd[0], fileno(stdin));  /* parent */
/* parent reads from pipe */
```

Want: “sort < file1 | uniq”




```
close(fd[1]); close(fd[0]);  /* parent */
```


Example : “sort < file1 | uniq”

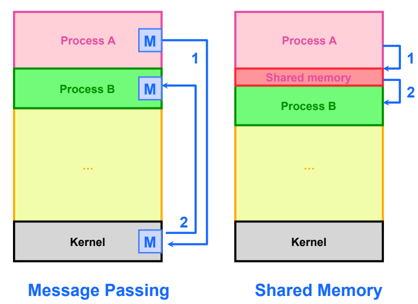
```
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <fcntl.h>

/* child | parent */
/* sort < file1.txt | uniq */
int main()
{
int status;
int fileDES;
int pipeDES[2];
pid_t pid;

fileDES = open( "myfile.txt", O_RDONLY );
dup2( fileDES, fileno( stdin) );

/* don't need to read via this one anymore */
close( fileDES ) ;

/* create a child that communicate via a pipe */
/* parent reads from pipe, child writes to pipe */
pipe( pipeDES );
```

```
pid = fork();
if( pid < 0 )
  {
  perror("fork");
  exit(1);
  }
else if( pid == 0 ) // child
  {
  close( pipeDES[0] );
  dup2( pipeDES[1], fileno(stdout) );
  close( pipeDES[1]);
  execl( "/usr/bin/sort", "sort", (char *) 0 );
  }
else if( pid > 0 ) // parent
  {
  close( pipeDES[1] );
  dup2( pipeDES[0], fileno(stdin) );
  close( pipeDES[0]);
  execl( "/usr/bin/uniq", "uniq", (char *) 0 );
  }
}
```


Communication Models

- Shared memory model
  - Share memory region for communication
  - Read and write data to shared region
  - Requires synchronization (e.g., locks)
  - faster
  - Setup time
- Message Passing model
  - Communication via exchanging messages

Communication Models



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - Example: Network processes (retrieval and analyzer) - one process takes stuff from the network and produce a package to be processed by another process (consumer)
  - *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

- Shared data:

```
#define BUFFER_SIZE 5
typedef struct
{
…
} item;
item buffer[BUFFER_SIZE];
int in = 0; /* first free item */
int out = 0; /* first full */
```

- If `in == out` empty
- If `(in + 1) % BUFFER_SIZE` full

Producer: `Insert()`

```
#define BUFFER_SIZE 5
typedef struct
{
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item nextProduced;

while (1)
  {
  while(((in + 1) % BUFFER_SIZE) == out )
    ; /* while full do nothing - wait */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  }
```


Consumer `Remove()`

```
#define BUFFER_SIZE 5
typedef struct
{
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item nextConsumed;

while(1)
  {
  while( in == out )
    ; /* do nothing */
  // remove an item from buffer/consume
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE
  }
```



```
item nextProduced;

while (1)
  {
  while(((in + 1) % BUFFER_SIZE) == out )
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  }
```

```
item nextConsumed;

while(1)
  {
  while( in == out )
    ; /* do nothing */
  // remove an item from buffer/consume
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE
  }
```

```
#define BUFFER_SIZE 5
typedef struct
{
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Message Passing Systems

- NO shared state
- `send()` & `receive()` primitives
- Processes communicate over *links*

Implementation Questions

- How are links established?
  - » Direct (explicitly name each other) or
  - » Indirect (mailboxes, ports)
- Can a link be associated with more than two processes?
  - » Symmetric/Asymmetric connections?
- Is a link unidirectional or bi-directional?
- Other Limits and Constraints:
  - » How many links can there be between every pair of communicating processes?
  - » What is the capacity of a link?
    - – Zero, bounded, unbounded: Explicit, or Automatic Buffering
  - » Can messages be of fixed or variable size ?

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.
  - » The link may be unidirectional, but is usually bi-directional.

Indirect Communication

- Messages are sent 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
  - » 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

Indirect Communication

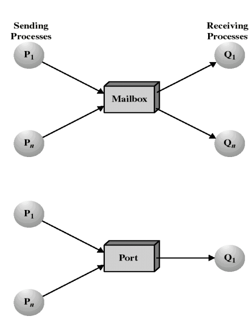
- Mailbox sharing
  - » P_1, P_2, and P_3 share mailbox A.
  - » P_1, sends; P_2 and P_3 receive.
  - » Who gets the message?
- Solutions
  - » Allow a link to be associated with at most two processes.
  - » Allow only one process at a time to execute a receive operation.
  - » Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Ownership of ports and mailboxes

- A port is usually own and created by the receiving process.
- The port is destroyed when the receiver terminates.
- The OS creates a mailbox on behalf of a process (which becomes the owner).
- The mailbox is destroyed at the owner's request or when the owner terminates.

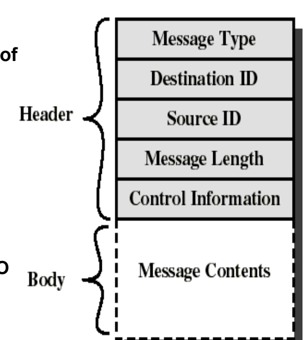
Mailboxes and Ports

- A mailbox can be private to one sender/receiver pair.
- The same mailbox can be shared among several senders and receivers:
  - » the OS may then allow the use of message types (for selection).
- Port: is a mailbox associated with one receiver and multiple senders
  - » used for client/server applications: the receiver is the server.



Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
  - » what to do if run out of buffer space.
  - » sequence numbers.
  - » priority.
- Queuing discipline: usually FIFO but can also include priorities.



Communication: Asynchronous or Synchronous

- Concerns the timing of corresponding operations
  - » e.g., in message passing how the timing of send and receives are coordinated.
- Synchronous Communication
  - » Sender does not return until the matching receive has been posted on the destinations process.
- Asynchronous Communication
  - » No coordination between sender and receiver, a message can be sent or received at any time without waiting for the receiver program to receive.
  - » Allows more concurrency
  - » No synchronization between the sender and the receiver
  - » Example: sender gets control back before the message has been copied or sent.

Communication: Blocking or Non-Blocking

- Pertains to the behavior of the operation itself (e.g. send and receives)
- Blocking operations: the completion of the call is dependent on certain events.
- Non-blocking operations: the call return without waiting for any event to complete (such as copying a message from user memory to system memory).
- Synchronous communication is often implemented using blocking operators and asynchronous communication using non-blocking operators.

Buffering

- Queue of messages attached to link:
  - » Zero capacity
    - – 0 message - link cannot have any messages waiting
    - – Sender must wait for receiver (rendezvous)
  - » Bounded capacity
    - – n messages - finite capacity of n messages
    - – Sender must wait if link is full
  - » Unbounded capacity
    - – infinite messages -
    - – Sender never waits

Client-Server Communication

- Remote Procedure Calls
- Remote Method Invocation (Java)
- Socket communication

Remote Procedure Calls (RPC)

- Inter-machine process to process communication
  - » Abstract procedure calls over a network:
  - » Rusers, rstat, rlogin, rup => deamons
  - » Hide message passing I/O from programmer
- Looks (almost) like a procedure call -- but client invokes a procedure on a server.
  - » Pass arguments – get results
  - » Fits into high-level programming languages
  - » Well understood

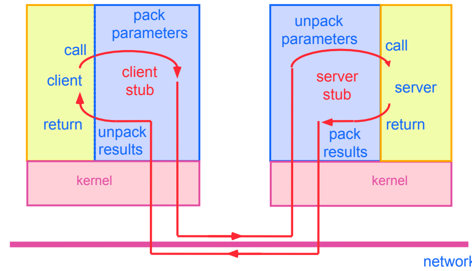
Remote Procedure Calls (RPC)

- RPC High level view:
  - » Calling process (client) is suspended
  - » Parameters are passed across network to a process server
  - » Server executes procedure
  - » Return results across network
  - » Calling process resumes

Remote Procedure Calls

- Usually built on top sockets (IPC)
- 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.

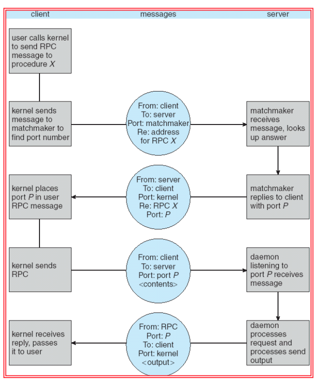
Client/Server Model Using RPC



Each RPC invocation by a client process calls a ***client stub***, which builds a message and sends it to a ***server stub***

- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

Execution of RPC

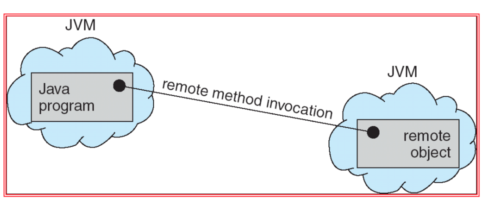


Remote Procedure Calls

- Machine independent representation of data:
  - » Differ if most/least significant byte is in the high memory address
  - » External data representation (XDR)
- Fixed or dynamic address binding
  - » Dynamic: Matchmaker daemon at a fixed address (given name of RPC returns port of requested daemon)

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.
- Possible to Pass Objects(remote, local) as parameters to remote methods (via serialization).



Marshalling Parameters

- Client invoke method: someMethod on a remote object Server