Processes

Yi Shi
Fall 2018
Xi'an Jiaotong University

Review: Protecting Processes from Each Other

- Problem: multiplexing resources
  - Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation (base/limit registers, page tables, etc)
  - Dual Mode Operation
  - Privileged instructions (set timer, I/O, etc)
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Review: OS Structure

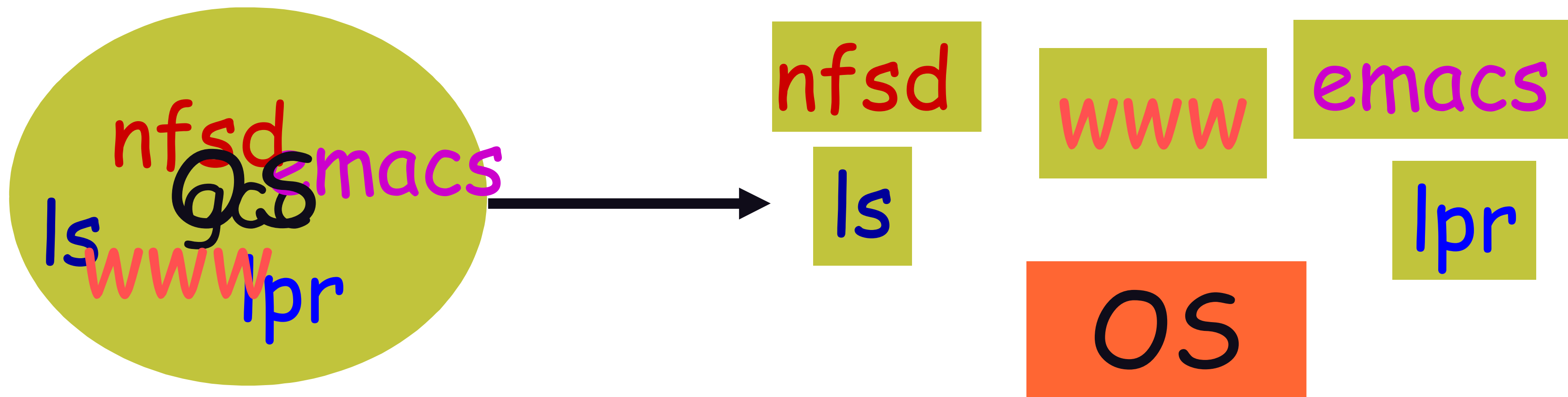
- Monolithic
  - Advantages: performance
  - Disadvantages: difficult to extend, debug, secure, and make reliable
- Layered
  - Advantages: simplicity of construction, debugging, and extensible
  - Disadvantages: defining layers, performance overhead
- Micro-kernel
  - Advantages: easy to extend, port. More reliable and secure.
  - Disadvantage: performance overhead
- Modular
  - Advantages: monolithic performance with layered flexibility
  - Disadvantages: modules can still crash system
- Virtual Machines
  - Advantages: protection/isolation, great systems building tool
  - Disadvantage: difficult to implement

Goals for Today

- What are processes?
  - Differences between processes and programs
- Creating and running a program
- Process details
  - States
  - Data structures
  - Creating new processes
  - Process Termination
- Inter-process communication

Why Processes? Simplicity + Speed

- Hundreds of things going on in the system



- How to make things simple?
  - Decomposition
  - Separate each in an isolated process
- How to speed-up?
  - Overlap I/O bursts of one process with CPU bursts of another

What is a process?

- A program becomes a process when an executable file is loaded into memory
- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- The unit of execution
- The unit of scheduling
- Thread of execution + address space
- Is a program in execution
  - Sequential, instruction-at-a-time execution of a program.

The same as “job” or “task” or “sequential process”

What is a program?

A program consists of:

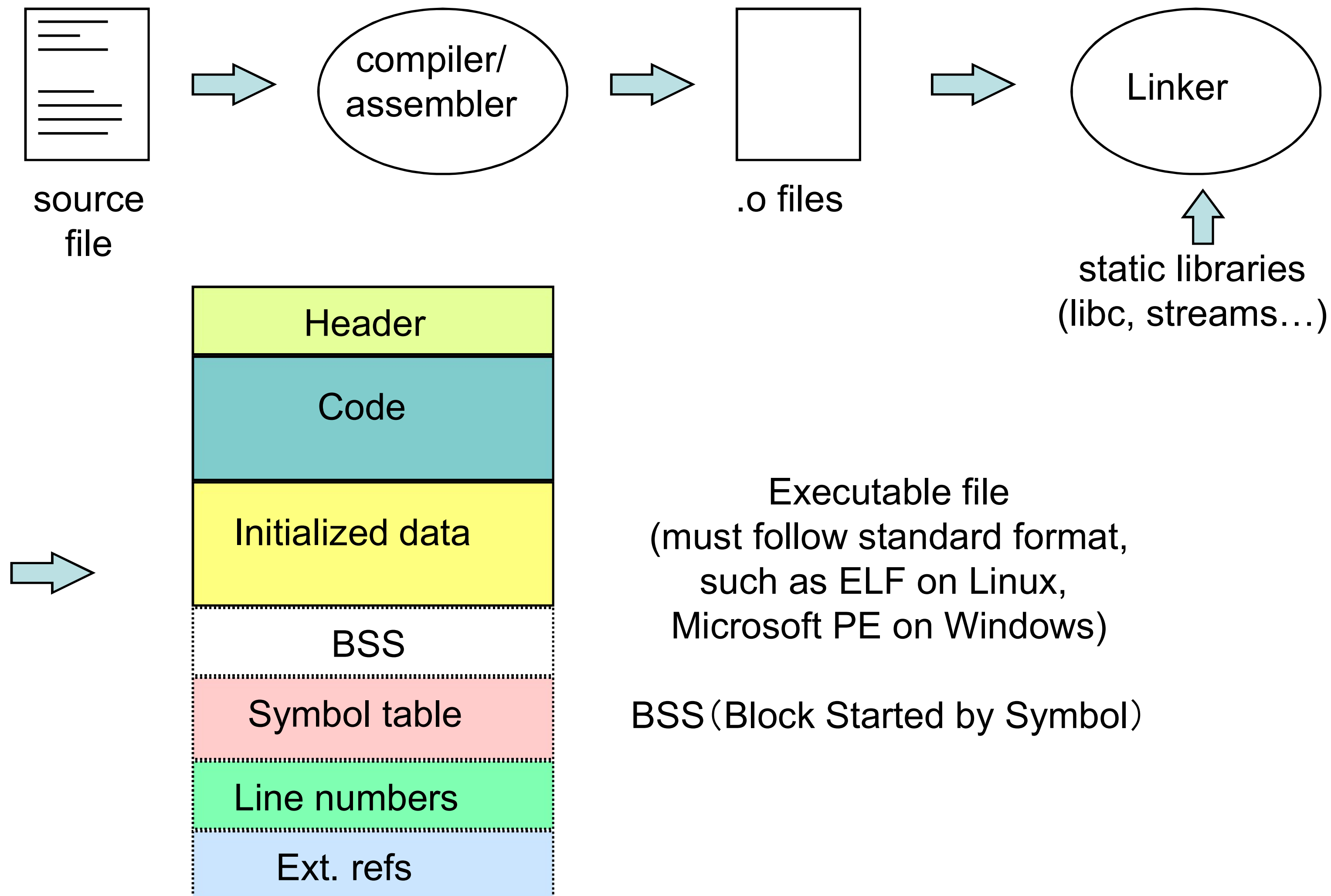
- **Code:** machine instructions
- **Data:** variables stored and manipulated in memory
  - initialized variables (globals)
  - dynamically allocated variables (malloc, new)
  - stack variables (C automatic variables, function arguments)
- **DLLs:** libraries that were not compiled or linked with the program
  - containing code & data, possibly shared with other programs
- **mapped files:** memory segments containing variables (mmap())
  - used frequently in database programs

- What's the relationship between a program and process?
  - *A process is an executing program*

Goals for Today

- What are processes?
  - Differences between processes and programs
- Creating and running a program
- Process details
  - States
  - Data structures
  - Creating new processes
  - Process Termination
- Inter-process communication

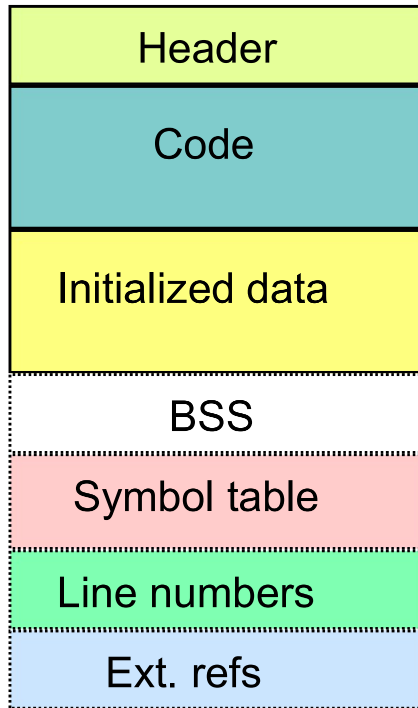
Preparing a Program



Running a program

- OS creates a “process” and allocates memory for it
- The loader:
  - reads and interprets the executable file
  - sets process’s memory to contain code & data from executable
  - pushes “argc”, “argv”, “envp” on the stack
  - sets the CPU registers properly & calls “__start()” [Part of CRT0]
- Program start running at __start(), which calls main()
  - we say “process” is running, and no longer think of “program”
- When main() returns, CRT0 calls “exit()”
  - destroys the process and returns all resources

Process != Program



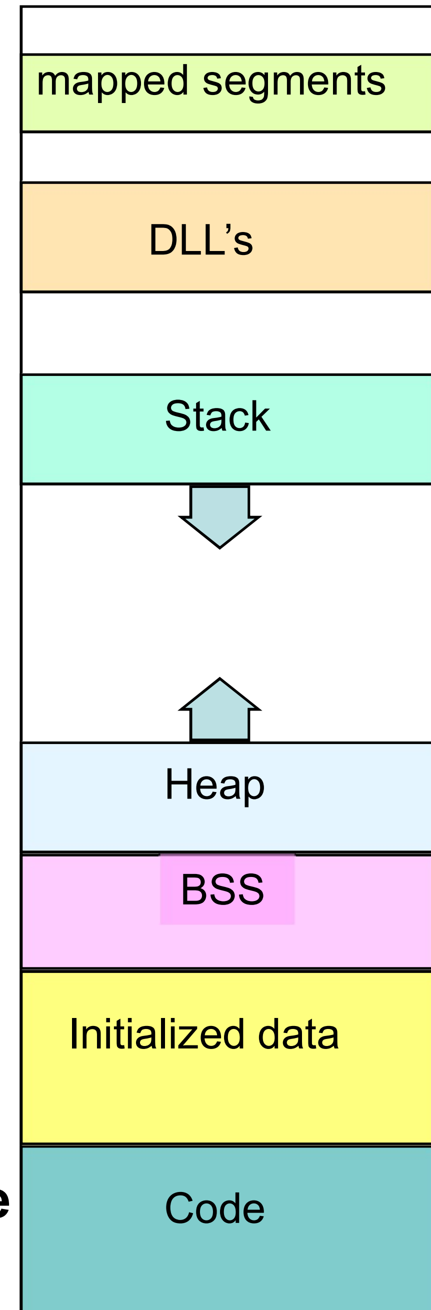
Executable

Program is passive
• Code + data

Process is running program
• stack, regs, program counter

Example:
We both run IE:
- Same program
- Separate processes

Process address space



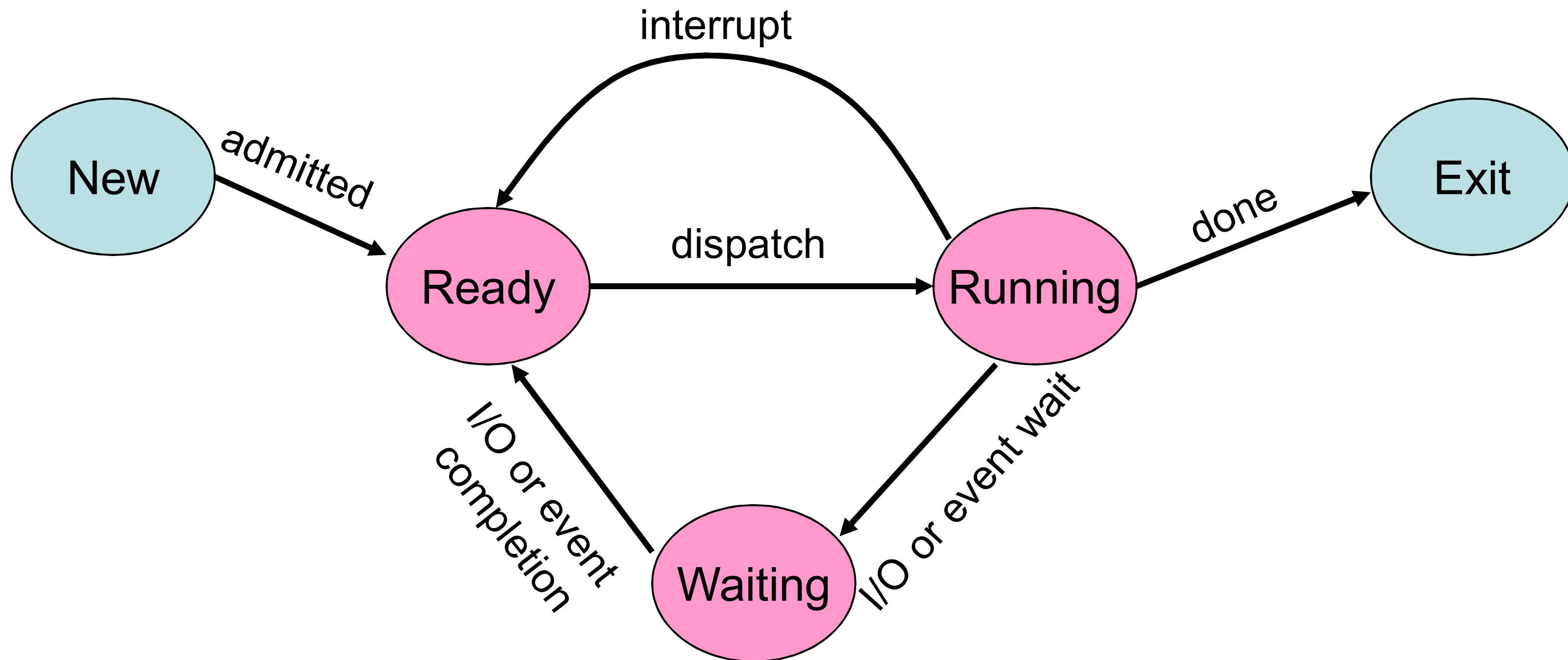
Goals for Today

- What are processes?
  - Differences between processes and programs
- Creating and running a program
- Process details
  - States
  - Data structures
  - Creating new processes
  - Process Termination
- Inter-process communication

Process States

- Many processes in system, only one on CPU
- "Execution State" of a process:
  - Indicates what it is doing
  - Basically 3 states:
    - Ready: waiting to be assigned to the CPU
    - Running: executing instructions on the CPU
    - Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

Process State Transitions



Processes hop across states as a result of:

- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

Process Data Structures

- OS represents a process using a *PCB*
  - Process Control Block
  - Has all the details of a process

Process Id	Security Credentials
Process State	Username of owner
General Purpose Registers	Queue Pointers
Stack Pointer	Signal Masks
Program Counter	Memory Management
Accounting Info	…

Context Switch

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process

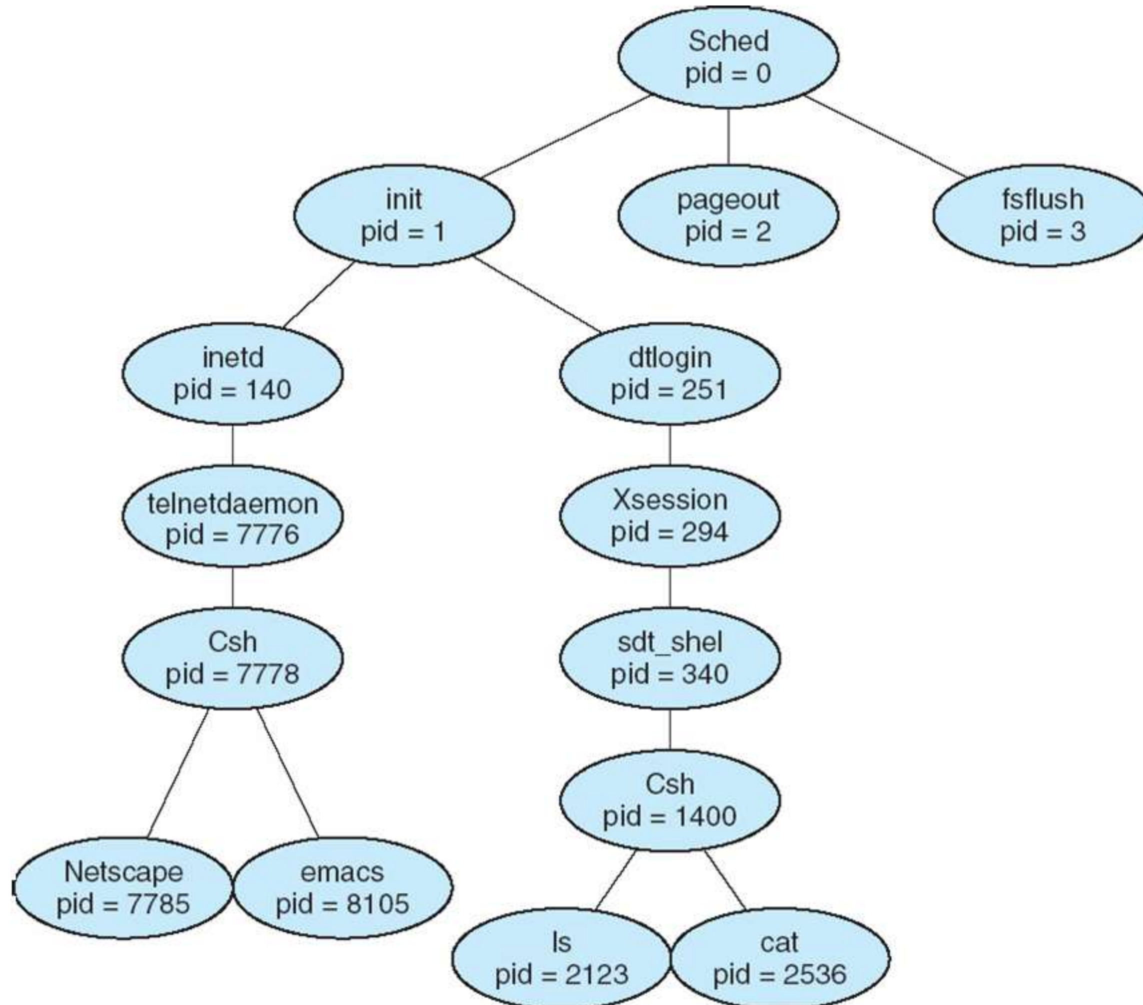
$\Rightarrow$ **Context Switch**

  - Process of switching CPU from one process to another
  - Very machine dependent for types of registers

How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - E.g. *sched* for Solaris, *ntoskrnel.exe* for XP
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships is expressed by a process tree
- For example, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (#processes, priorities…)
- *Explorer.exe* in Windows for graphical interface

How to create a process?



Processes Under UNIX

- Fork() system call is only way to create a new process
- int fork() does many things at once:
  - creates a new address space (called the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

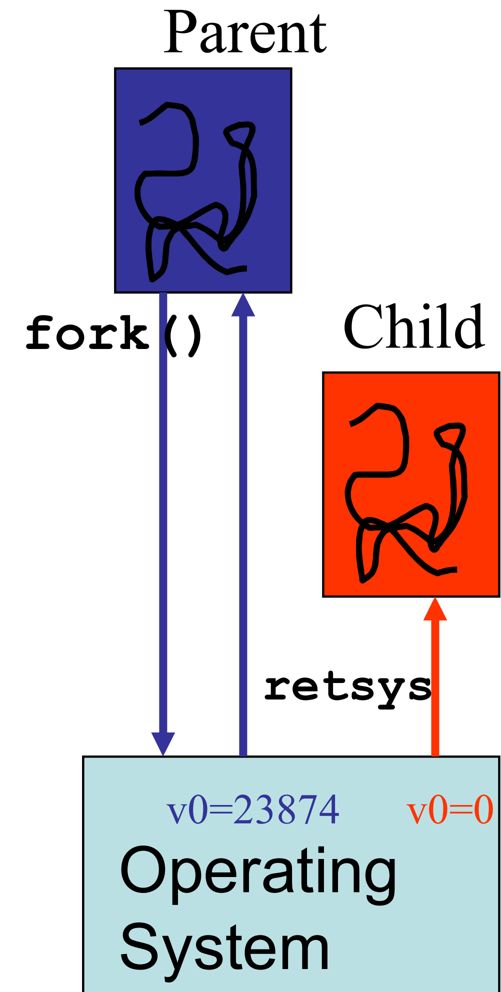
Example

```
main(int argc, char **argv)
{
    char *myName = argv[1];
    int cpid = fork();
    if (cpid == 0) {
        printf("The child of %s is %d\n", myName, getpid());
        exit(0);
    } else {
        printf("My child is %d\n", cpid);
        exit(0);
    }
}
```

What does this program print?

Bizarre But Real

```
lace:tmp<15> cc a.c
lace:tmp<16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Fork is half the story

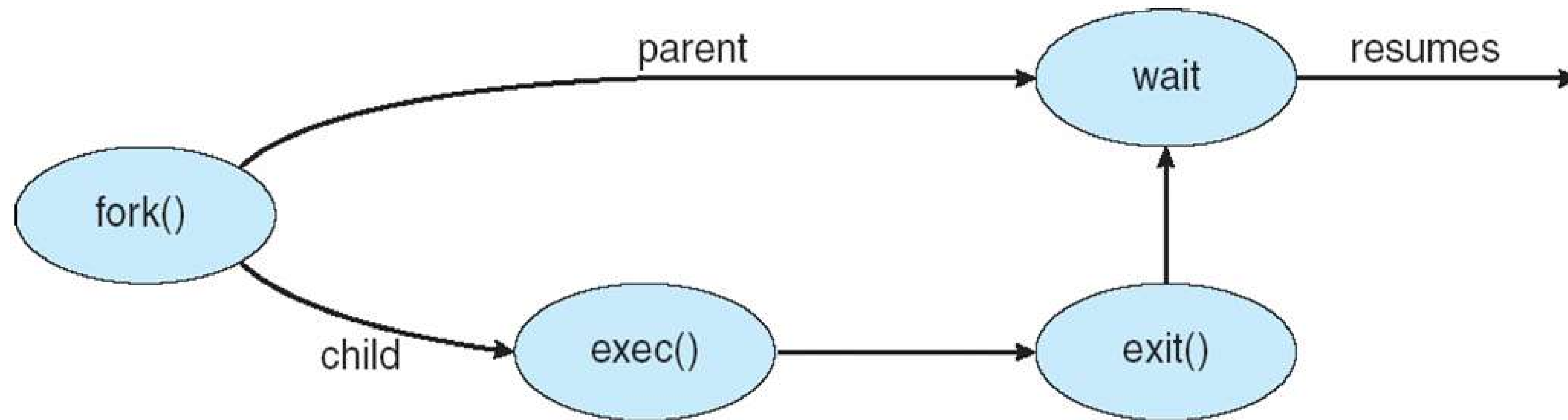
- Fork() gets us a new address space,
  - but parent and child share EVERYTHING
    - memory, operating system state
- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by programName
  - starts executing at header.startPC
  - Does not return
- Pros: Clean, simple
- Con: duplicate operations

Starting a new program

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   char *progName = argv[2];

   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      execlp("/bin/ls",    // executable name
             "ls", NULL); // null terminated argv
} else {
      wait(NULL);
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

Process Creation



Process Termination

- Process executes last statement and OS decides(**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of child process (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OSes don't allow child to continue if parent terminates(VMS)
      - All children terminated - *cascading termination*

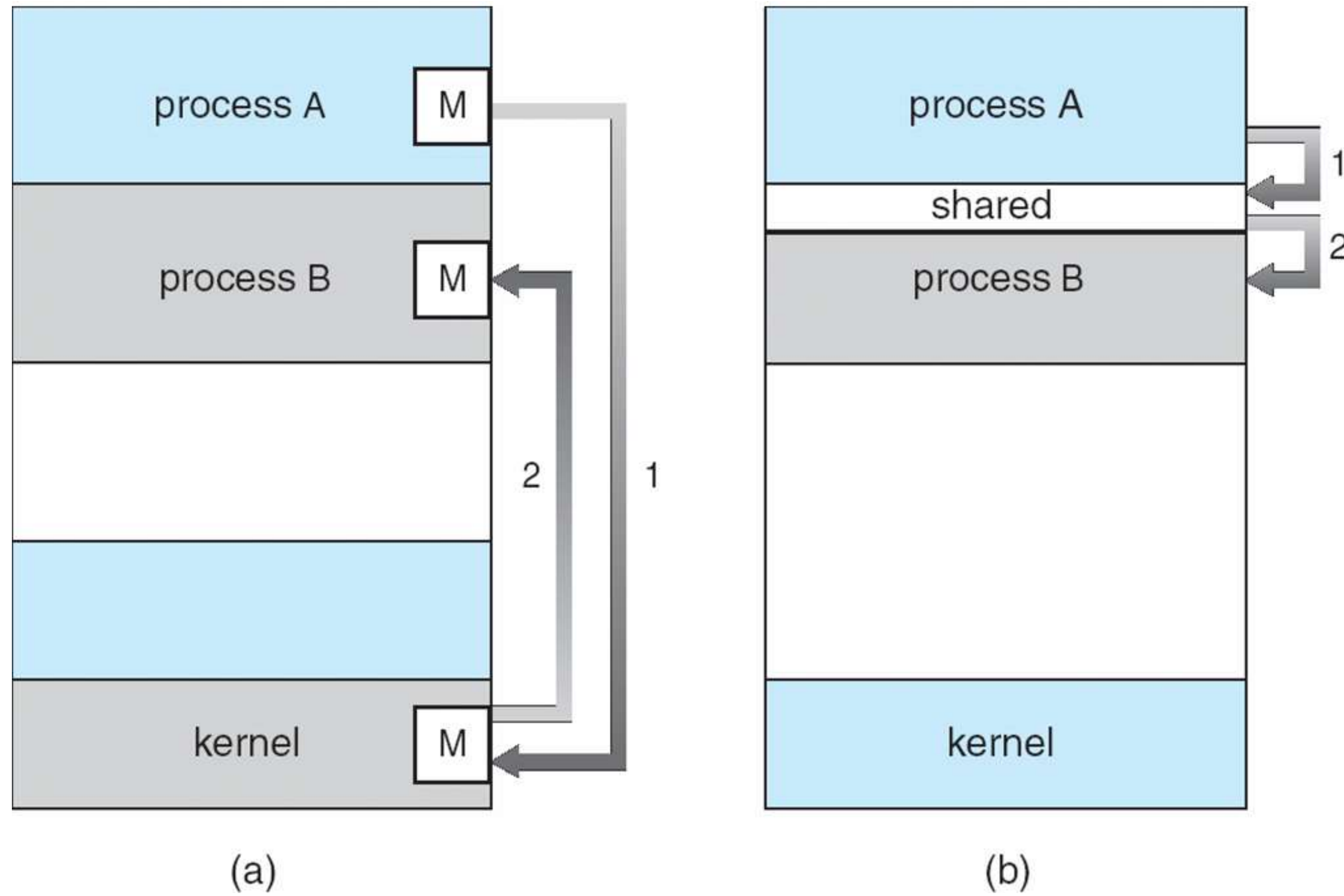
Goals for Today

- What are processes?
  - Differences between processes and programs
- Creating and running a program
- Process details
  - States
  - Data structures
  - Creating new processes
  - Process Termination
- Inter-process communication (IPC)

Why IPC?

- Independent vs Cooperating processes
- Why let processes cooperate?
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience: a user may work on many tasks
- Cooperating processes need IPC
- Two fundamental models
  - Message Passing: easier, smaller amounts of data, slower
  - Shared Memory: harder, large amounts of data, maximum speed

Communication Models



Shared Memory

- Processes establish a segment of memory as shared
  - Typically part of the memory of the process creating the shared memory. Other processes attach this to their memory space.
- Requires processes to agree to remove memory protection for the shared section
  - Recall that OS normally protects processes from writing in each others memory.

Producer/Consumer using shared memory

- *Producer* process produces information consumed by *Consumer* process.
  - Very common paradigm.

```
#define BUFFER_SIZE 10

typedef struct{
   ..some stuff..
}item;

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

Producer/Consumer (1/2)

- Producer process:

```
item nextProduced;

while(true)
{
   /*Produce an item in next produced*/
   while(((in + 1) % BUFFER_SIZE) == out);
      //do nothing…
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Producer/Consumer (2/2)

- Consumer process

```
item nextConsumed;

while(true)
{
    while(in == out);
        //do nothing..
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* Consume item in nextConsumed */
}
```

Synchronization

- The previous code only allows BUFFER_SIZE-1 items at the same time
- To remedy this, the processes would need to *synchronize* their access to the buffer. (This is a large topic, later).

Example

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

main(int argc, char **argv)
{
   char* shared_memory;
   const int size = 4096;
   int segment_id = shmget(IPC_PRIVATE, size, S_IRUSR |S_IWUSR);
   int cpid = fork();
   if (cpid == 0)
   {
      shared_memory = (char*) shmat(segment_id, NULL, 0);
      sprintf(shared_memory, "Hi from process %d",getpid());
      shmdt(shared_memory);
   }
   else
   {
      wait(NULL);
      shared_memory = (char*) shmat(segment_id, NULL, 0);
      printf("Process %d read: %s\n", getpid(), shared_memory);
      shmdt(shared_memory);
      shmctl(segment_id, IPC_RMID, NULL);
   }
}
```

Message Passing

- Send(P, msg): Send msg to process P
- Recv(Q, msg): Receive msg from process Q
- Useful in in distributed environment and typically requires kernel intervention
- Naming:
  - Hardcode sender&receiver/direct communication (symmetry and asymmetry)
  - Indirection using mailboxes/ports

Synchronization

- Possible primitives:
  - Blocking send/receive
  - Non-blocking send/receive
  - Also known as *synchronous* and *asynchronous*.
- When both send and receive are blocking, we have a *rendezvous* between the processes. Other combinations need *buffering*.

Buffering

- Zero capacity buffer
  - Needs synchronous sender.
- Bounded capacity buffer
  - If the buffer is full, the sender blocks.
- Unbounded capacity buffer
  - The sender never blocks.

Summary

- The unit of execution and scheduling
- Thread of execution (next time) + address space
- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- Sequential, instruction-at-a-time execution of a program. *Operating system abstraction to represent what is needed to run a single, multithreaded program*
- Program != Process
  - A process is a program in execution
- Abstraction used for protection
  - Main Memory State (contents of Address Space)
- Multiprogramming: overlap IO and CPU
- Context Switches are expensive