Reading Reference

1. Textbook Chapters 2 and 3
2. Molay Reference Text: Chapter 8

WEEK 3 – UNIX PROCESS

CSCE 313 Spring 2017

Key Learnings from Week 2

- DUAL MODE
  - "Referee" Role of an OS comes with significant responsibilities and capabilities
    - Enforcing Fairness, Efficiency, and Correctness are perhaps the most critical of the bunch
    - OS is also a piece of software residing in the same memory so it is the CPU that wears the "**referee**" hat when it runs the **OS Kernel** code and wears the "**player**" hat when running **user code**
    - This is called **Dual Mode operation**

Key Learnings from Week 2

- **Architectural support** for user and kernel modes in CPU execution implies **hardware features** provided to accomplish dual modes transition, especially
  - Privileged Instructions
  - Memory Protection
  - Timer, etc.

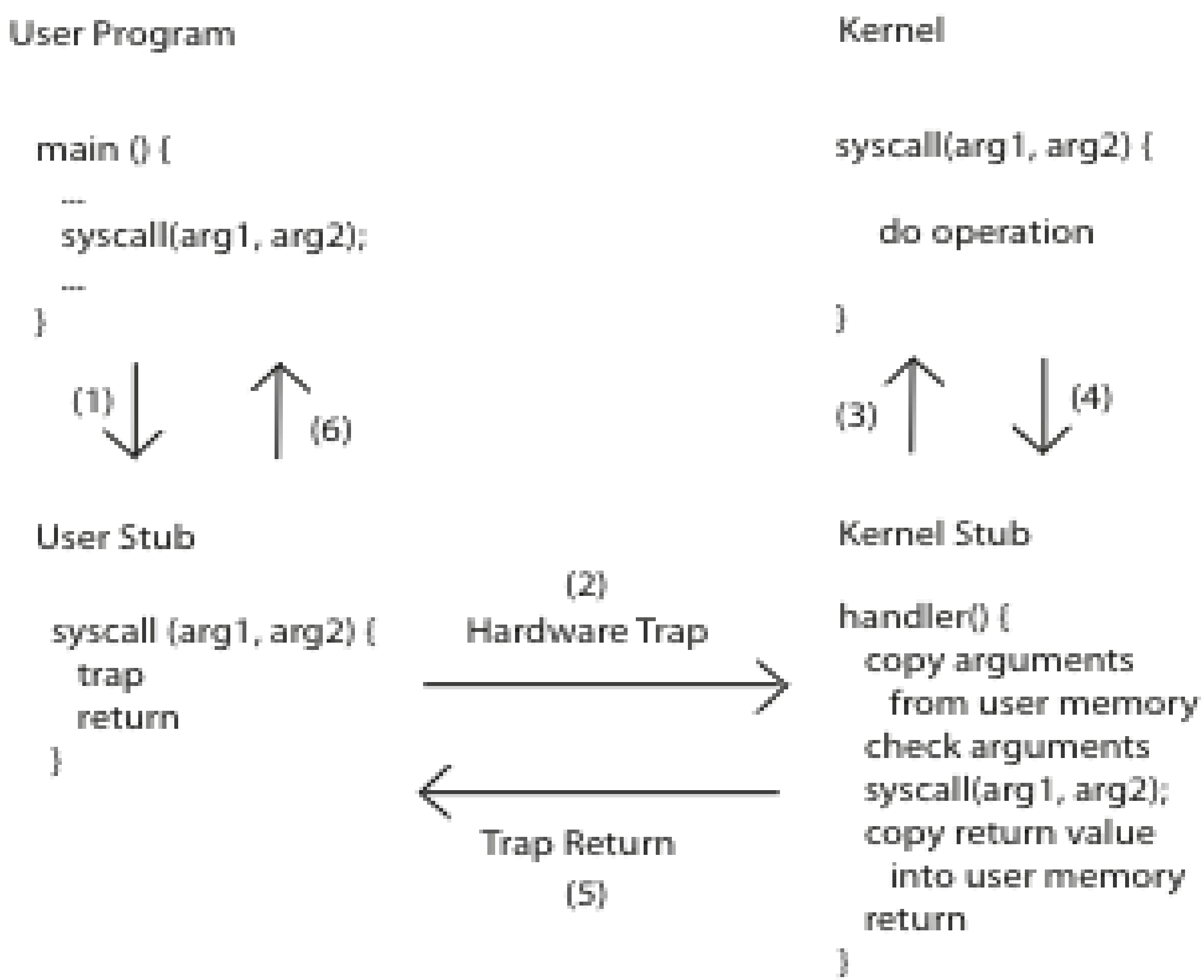
A Real-Life Analogy (Approximate)

A Typical Coffee Shop	Computer System
Store	System
Customer	Process or Program or User Application
Barista/Cashier	Operating System Kernel, Privileged Code
Coffee Machine	CPU
Customer Order	System Call
Order item not on Menu	Exception
Telephone Call	Interrupt
Fire Alarm	Signal
>1 Customers being served	Process Scheduling
Customer realizing at the counter that he needs to go to ATM to get money	Process Context Switching

Key Asides

- System Call Handling (Ch. 2.6)
  - How do we execute traps safely and return back cleanly to resume user code?
- Interrupt handling (Ch. 2.5)
  - How do we **service an exception** that occurs in the middle of running user code and **return back cleanly (safely)** to resume user code?
- How do we boot an OS Kernel? (Ch. 2.9)

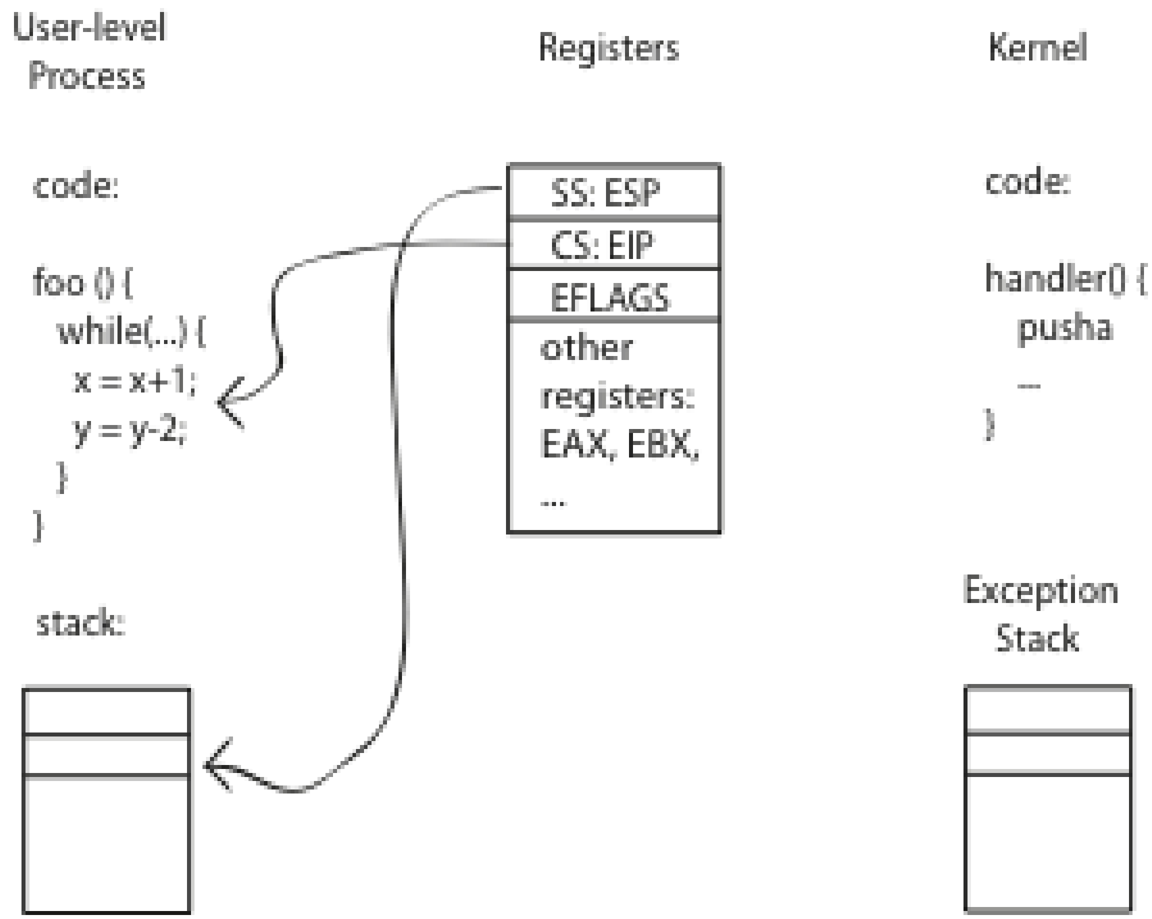
Aside1: Handling System Calls



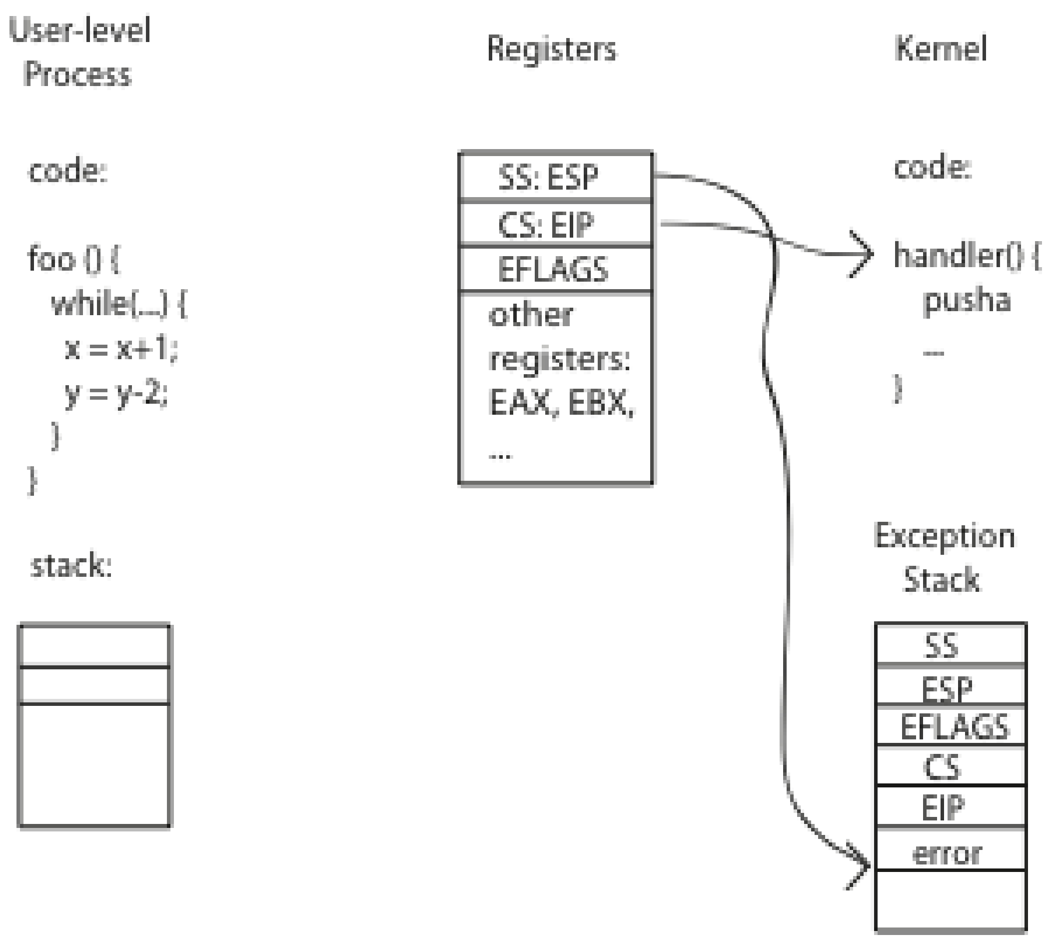
Aside2: Handling an Interrupt

A. CPU checks for interrupts after each instruction
D. Save critical registers on Kernel stack and get in Kernel Mode
B. Disable Interrupts
C. Refer to Interrupt Descriptor Table for **handler location**
E. Execute handler
   - save process context
   - service INT
F. Enable Interrupts. If no other INT, restore control to interrupted process
G. Continue on normal program execution

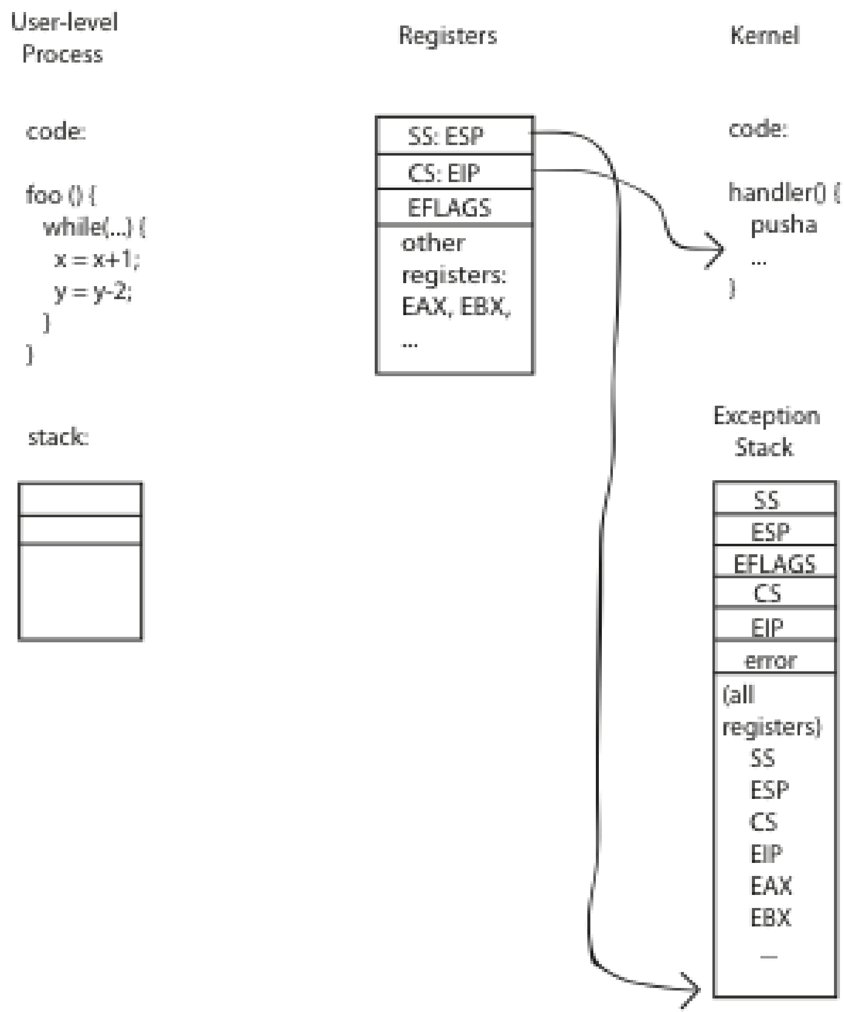
Aside2: Handling an Interrupt (Before)



Aside2: Handling an Interrupt (During)



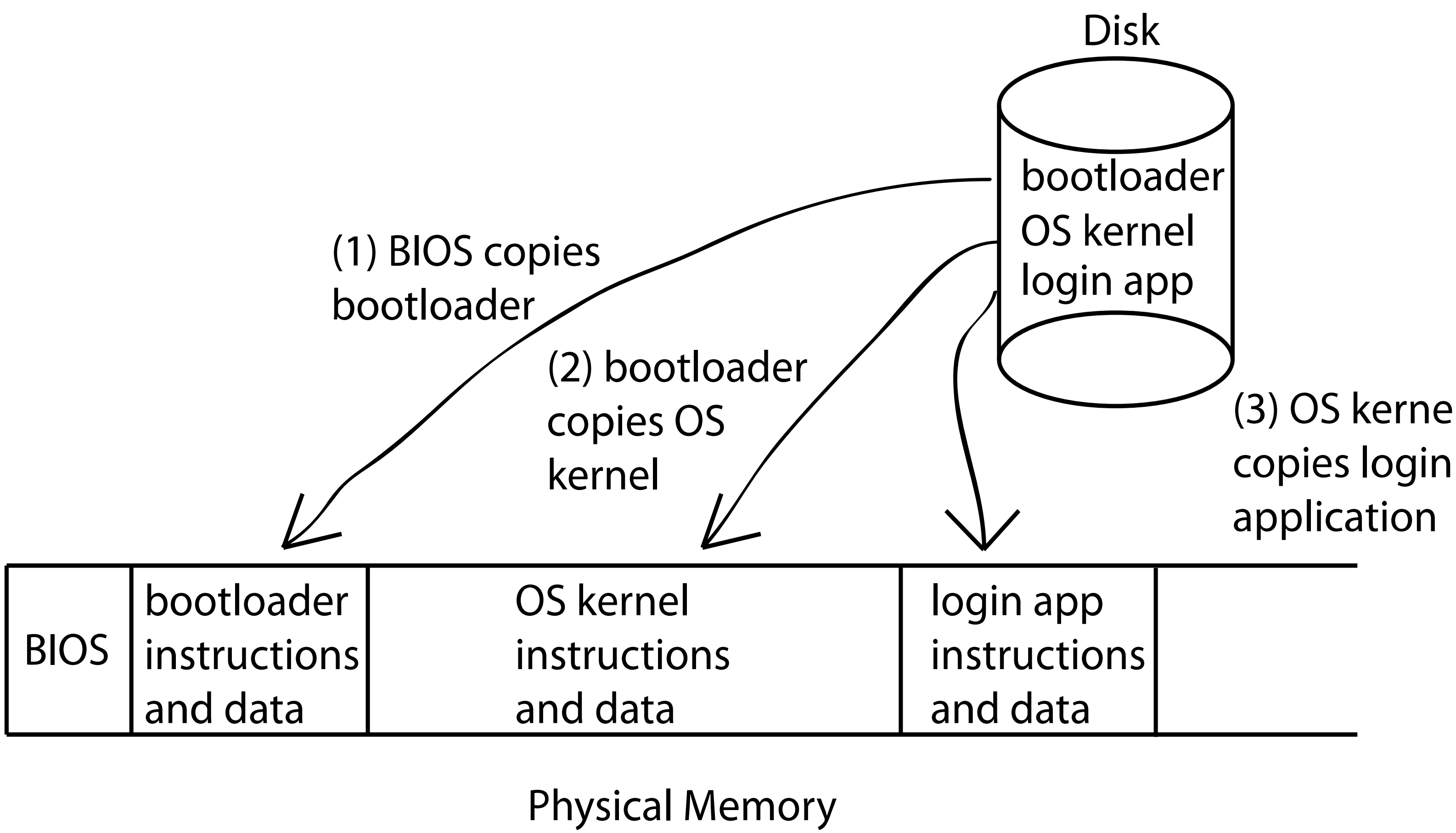
Aside2: Handling an Interrupt (After)



Aside2: At the end of handler

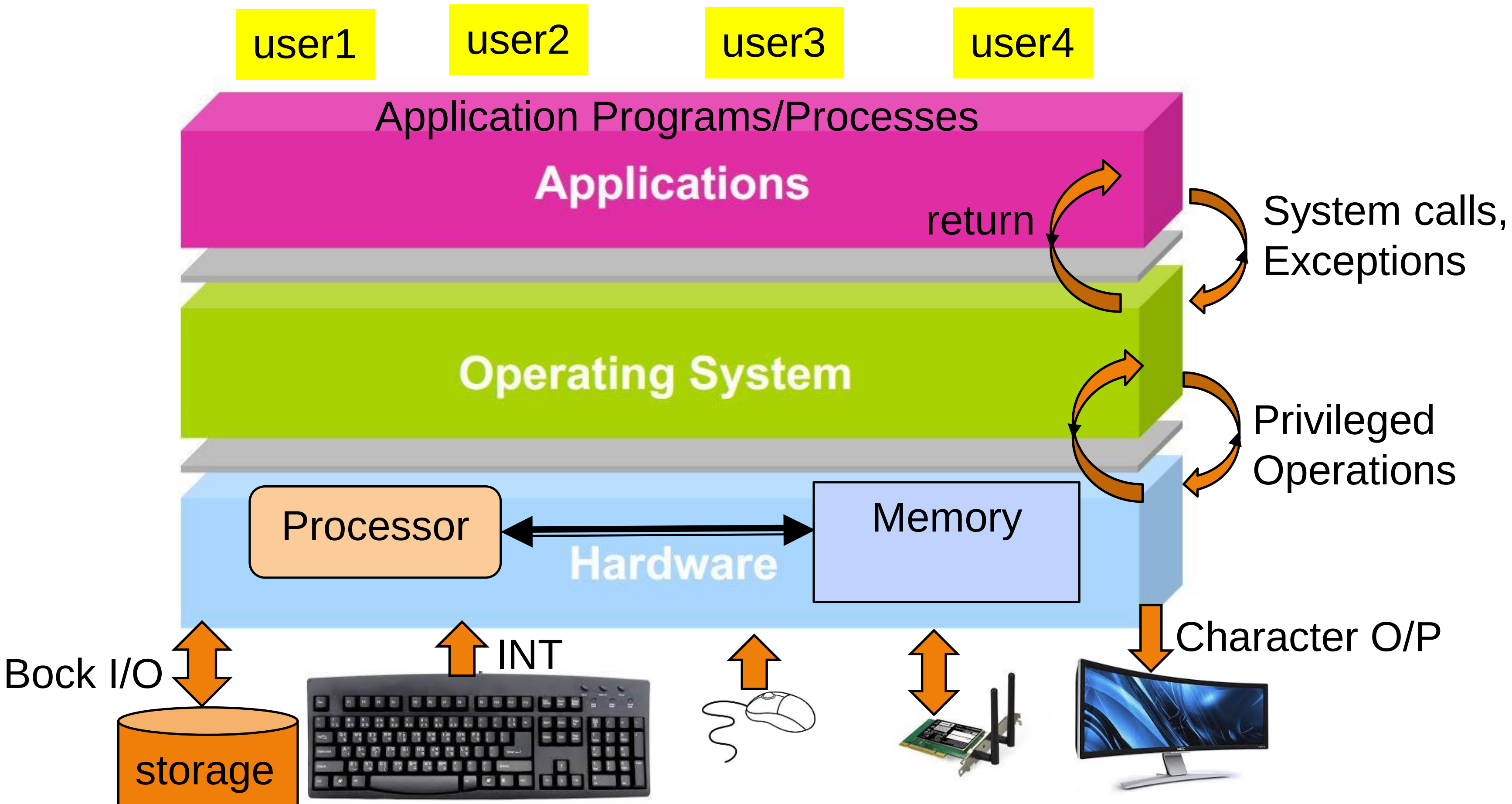
- Handler restores saved registers
- **Atomically** return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

Aside3: PC Booting



Theme of the rest of Week 3

- Unix Process concept and definitions



Outline

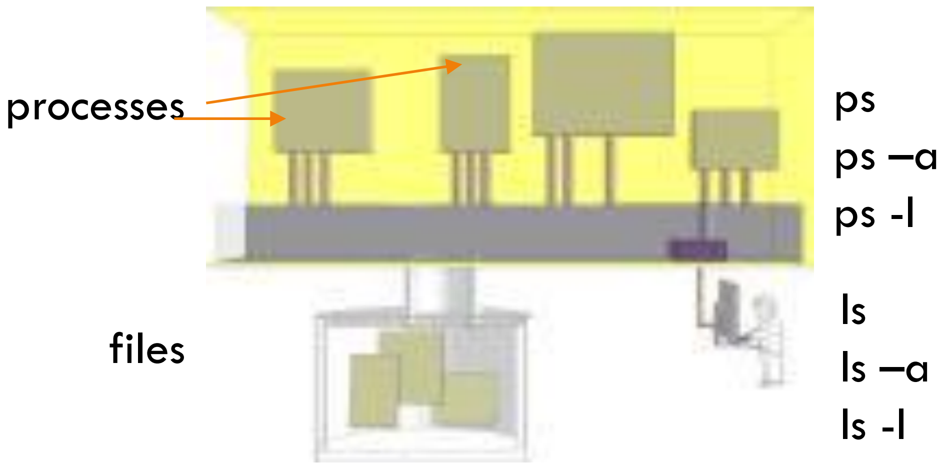
- Process – Program in Action
- Address Spaces
- Learning about Process with ‘ps’
- Miscellaneous questions about Process
- Concurrent Processes
- Context Switching

Prologue* with Questions

- *"How does Unix run programs? It looks easy enough: you log in, your shell prints a prompt, you type a command and press Enter. Soon a program runs.*
  - *When the program finishes, your shell prints a new prompt. How does that work?*
  - *What is the shell? What does a shell do? What does the kernel do? What is a program and what does it mean to run a program?"*

* Understanding Linux/Unix Programming, by Bruce Molay

Process is Program in Action



- In Unix terminology
  - an executable program is a list of machine language instructions and data
  - a process is the memory space and settings with which the program runs
- Data and programs are stored in files on the disk: programs run in processes

Learning about Processes with ‘ps’


```
[tyagi]@linux2 ~> (09:40:04 02/02/15)
:: ps
  PID TTY          TIME CMD
15038 pts/40   00:00:00 bash
15092 pts/40   00:00:00 ps
```

```
[tyagi]@linux2 ~> (10:42:46 09/15/15)
:: ps -la
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 36632  1006   940  0  80   0 -  3195 wait   pts/6    00:00:00 sh
0 R 36632  1012  1006 99  80   0 -  2947 ?      pts/6    03:03:44 my_allocator
4 S     0  4637  4607  0  80   0 -  9255 ?      pts/0    00:00:00 sudo
4 S     0  4638  4637  0  80   0 - 22611 wait   pts/0    00:00:00 su
4 S     0  4639  4638  0  80   0 -  3638 ?      pts/0    00:00:00 bash
4 S     0  6854  6721  0  80   0 -  9255 ?      pts/1    00:00:00 sudo
4 S     0  6855  6854  0  80   0 - 22611 wait   pts/1    00:00:00 su
4 S     0  6856  6855  0  80   0 -  3604 ?      pts/1    00:00:00 bash
0 S 38172 21808 19088  0  80   0 - 28939 futex_ pts/14   00:00:00 ghc
0 S 36917 21822 21510  0  80   0 - 29218 futex_ pts/25   00:00:00 ghc
0 S 35691 22101 17390  0  80   0 -  4960 ?      pts/17   00:00:00 vim
0 S 35691 22215 17426  0  80   0 -  4960 ?      pts/22   00:00:00 vim
0 R 38345 23383 2286
0 S 36662 31883 1332
0 S 36804 40135 39988  0  80   0 -  5696 pause  pts/2    00:00:00 screen
4 S     0 45183 45133  0  80   0 -  9255 ?      pts/9    00:00:00 sudo
4 S     0 45184 45183  0  80   0 - 22611 wait   pts/9    00:00:00 su
4 S     0 45185 45184  0  80   0 -  3637 ?      pts/9    00:00:00 bash
4 S     0 48950 48920  0  80   0 -  9255 ?      pts/13   00:00:00 sudo
4 S     0 48951 48950  0  80   0 - 22611 wait   pts/13   00:00:00 su
4 S     0 48952 48951  0  80   0 -  3729 ?      pts/13   00:00:00 bash
```

```
[tyagi]@linux2 ~> (09:40:07 02/02/15)
:: ps -a
  PID TTY          TIME CMD
 4633 pts/0    00:00:00 sudo
 4634 pts/0    00:00:00 su
 4635 pts/0    00:00:00 bash
 5612 pts/14   00:00:00 ghc
 8479 pts/7    00:00:00 vim
10943 pts/28   00:00:01 ghc
12185 pts/23   00:00:15 a.out
12239 pts/23   00:00:37 a.out
12402 pts/32   00:00:00 ghc
14197 pts/22   00:00:00 a.out
14411 pts/19   00:00:00 vim
15447 pts/20   00:00:00 a.out
15540 pts/40   00:00:00 ps
28496 pts/5    00:00:00 sudo
28497 pts/5    00:00:00 su
28498 pts/5    00:00:00 bash
```

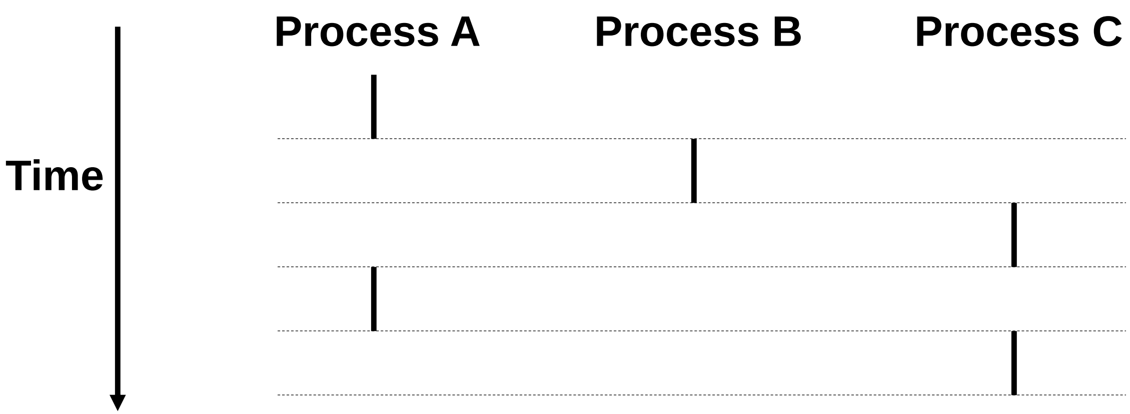
Run these commands in your linux/unix system and then also read the ‘man’ pages

Processes

- Definition: A *process* is an instance of a 'running' program
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private address space
    - Each program seems to have exclusive use of main memory
- How are these illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

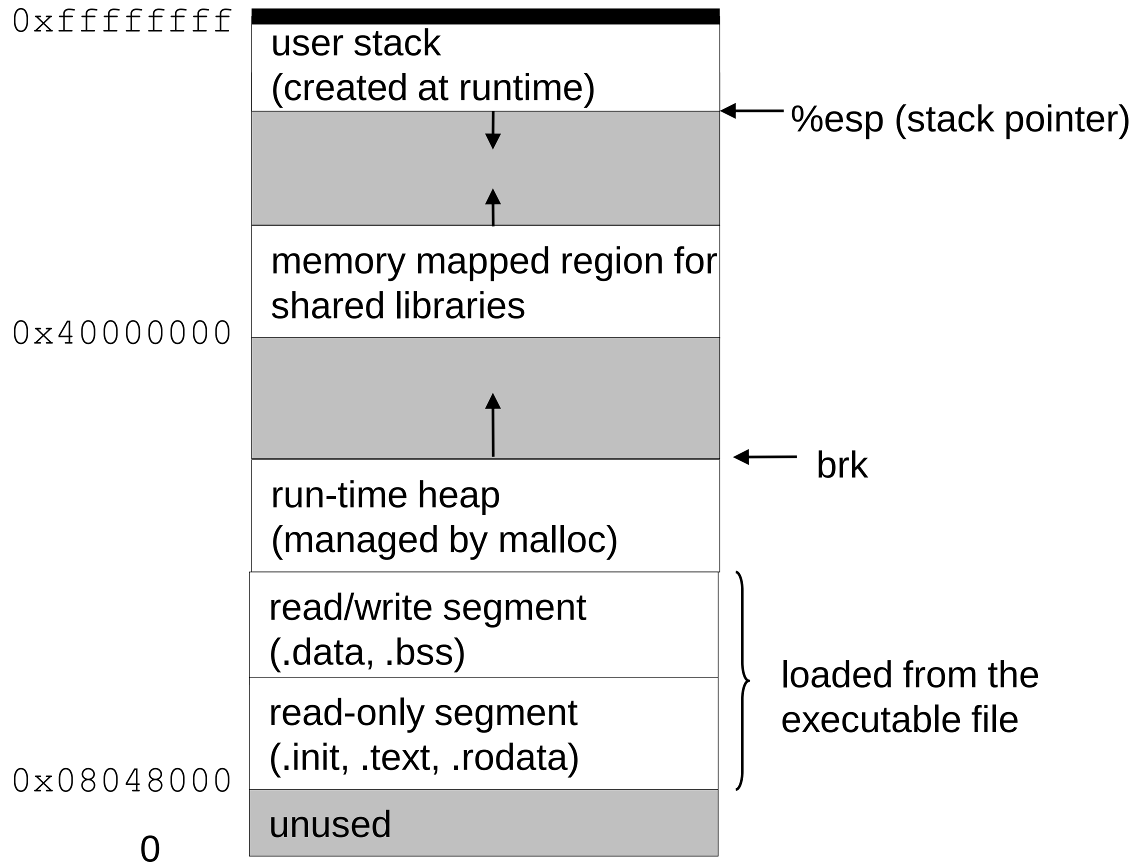
Logical Control Flows

Each process has its own logical control flow



Private Address Spaces

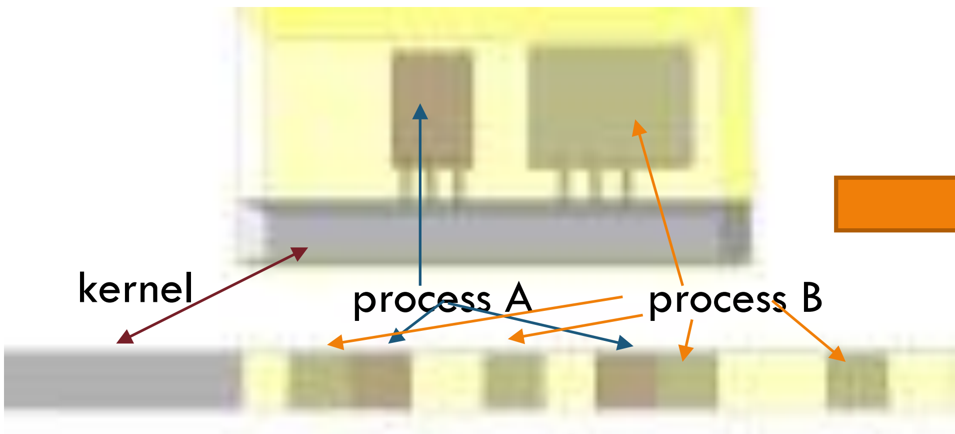
- Each process has its own private address space



Process Management and File Management

- ‘ps’ shows that processes have many attributes
- ‘ls’ does something similar but for files
- The kernel stores several processes in the memory just like it stores files on the disk

Computer Memory and Programs

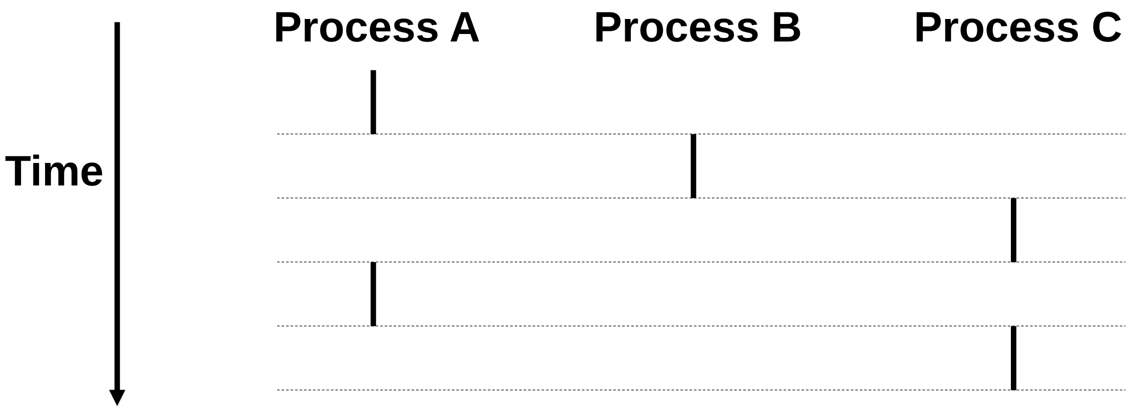


- Memory can be viewed as an expanse of space containing the kernel and user applications (processes)
- Memory as an array of pages and split processes into one or more pages

- The array of pages may be stored physically in solid state chips

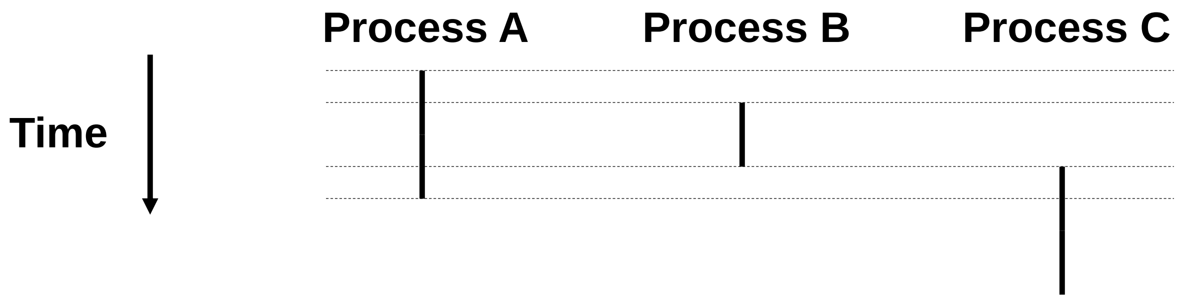
Concurrent Processes

- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples:
  - Concurrent: A & B, A & C
  - Sequential: B & C



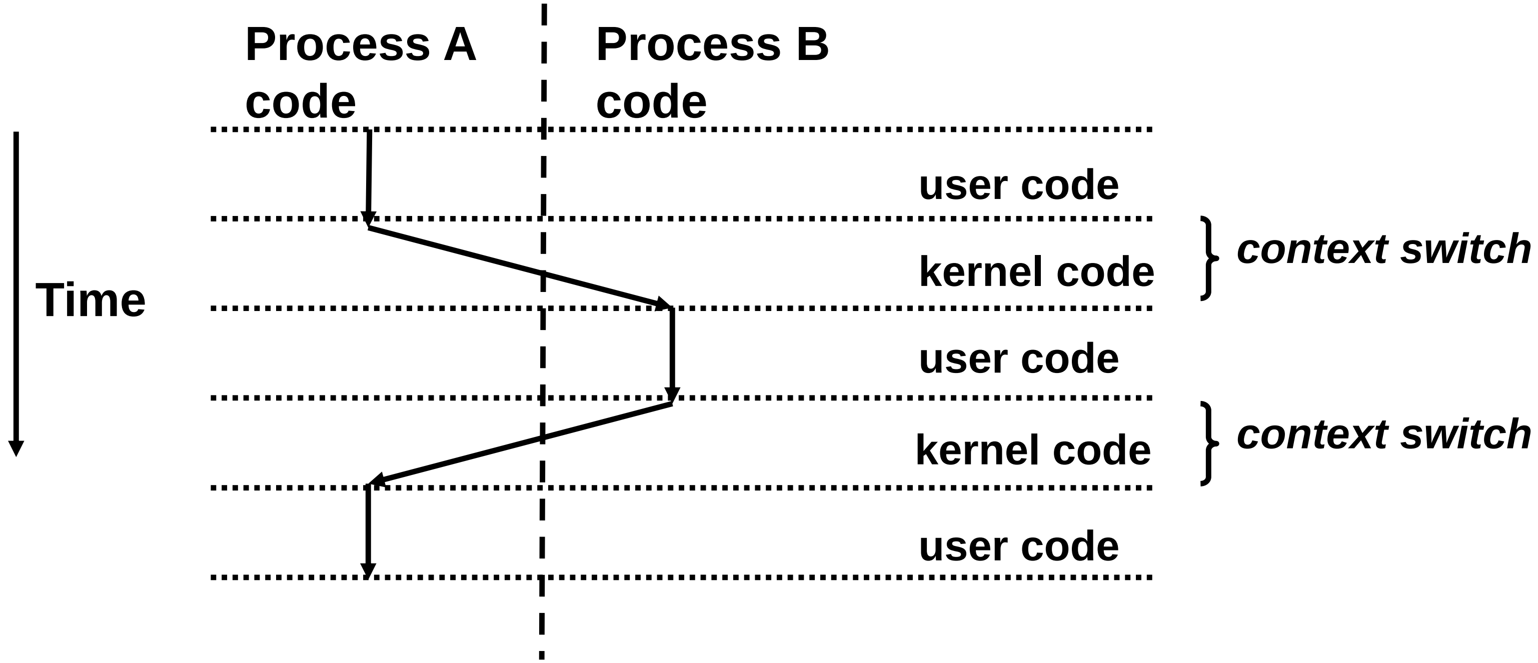
User View: Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time (except on multi-core machines)
- However, we can think of concurrent processes as running in 'parallel' with each other



Context Switching

- Processes are managed by the *kernel*
  - Important: the kernel runs as part of (or on behalf of) user processes
- Control flow passes from one process to another via a *context switch*



Some questions to ponder about processes

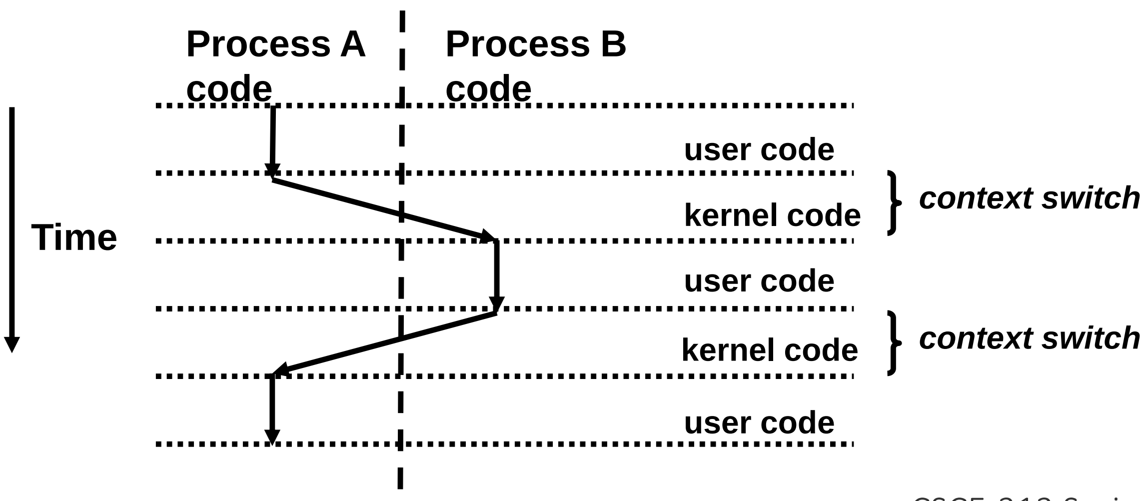
- How is a process created?
- How is a process deleted?
- Is there a user process and kernel process
- Where do we keep information about a process
- Does a process have to run through completion from start to finish or can it be interrupted?

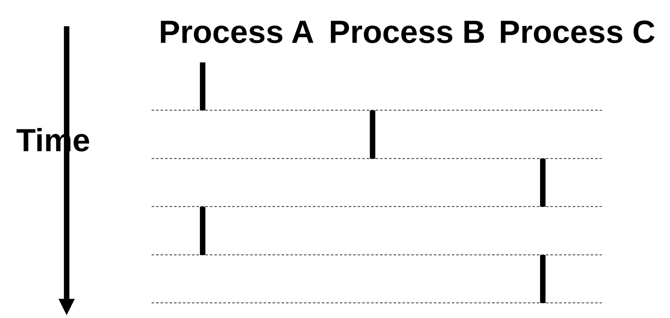
Some questions to ponder about processes

- Do processes have priorities?
- What are the relationships between multiple processes in a system?
- Can we have multiple processes related to the same program? Would multiple processes of the same program share addresses during execution?
- How does a program create and run a program?
- How does a parent wait for a child to exit?

OK, so what have we learnt so far…

- Concept and Definition of a Process
- Example viewed through UNIX ‘ps’
- Outlined some questions about processes for forthcoming discussions
- Process concurrency and context switching





What's coming up next?

- Process Operations and Programming Interface (Chapter 3)
- We will also start answering some of the questions posed earlier about a process
  - Executing a program from within a program. How does a shell work?
  - Creating a new process
  - Introducing Wait dependencies between parent and child processes

What is a Shell?

- Shell is a program which
  - Runs programs
  - Manages inputs and outputs
  - Can be programmed

Shell – Running Programs

- The commands ls, grep, date, etc. are regular programs. The shell loads these programs into memory and runs them.

```
[tyagi]@linux2 ~> (21:12:39 02/08/16)
:: ls
csce312  csce313  mybin  play

[tyagi]@linux2 ~> (21:12:47 02/08/16)
:: ls | grep bin
mybin

[tyagi]@linux2 ~> (21:12:58 02/08/16)
:: ls csce313/* > foo

[tyagi]@linux2 ~> (21:13:09 02/08/16)
:: TZ=PST8PDT; export TZ; date; TZ=CST6CDT
Mon Feb  8 19:14:02 PST 2016

[tyagi]@linux2 ~> (19:14:02 02/08/16)
:: date
Mon Feb  8 21:14:09 CST 2016
```

Shell – Managing I/O

- Using '>', '|' etc. the user tells the shell to attach the output to a file on disk, or to another process, etc.

```
[tyagi]@linux2 ~> (21:19:21 02/08/16)
:: whoami > myname

[tyagi]@linux2 ~> (21:19:28 02/08/16)
:: ls
csce312  csce313  foo  mybin  myname  play

[tyagi]@linux2 ~> (21:19:37 02/08/16)
:: cat myname
tyagi
```

Shell - Programming

- Shell is also a programming language with variables and flow control

```
[tyagi]@linux2 ~> (09:57:51 02/09/16)
:: NAME=tyagi

[tyagi]@linux2 ~> (09:59:43 02/09/16)
:: whoami > myname

[tyagi]@linux2 ~> (09:59:49 02/09/16)
:: if grep $NAME myname; then echo hello $NAME; fi
tyagi
hello tyagi

[tyagi]@linux2 ~> (09:59:53 02/09/16)
::
```

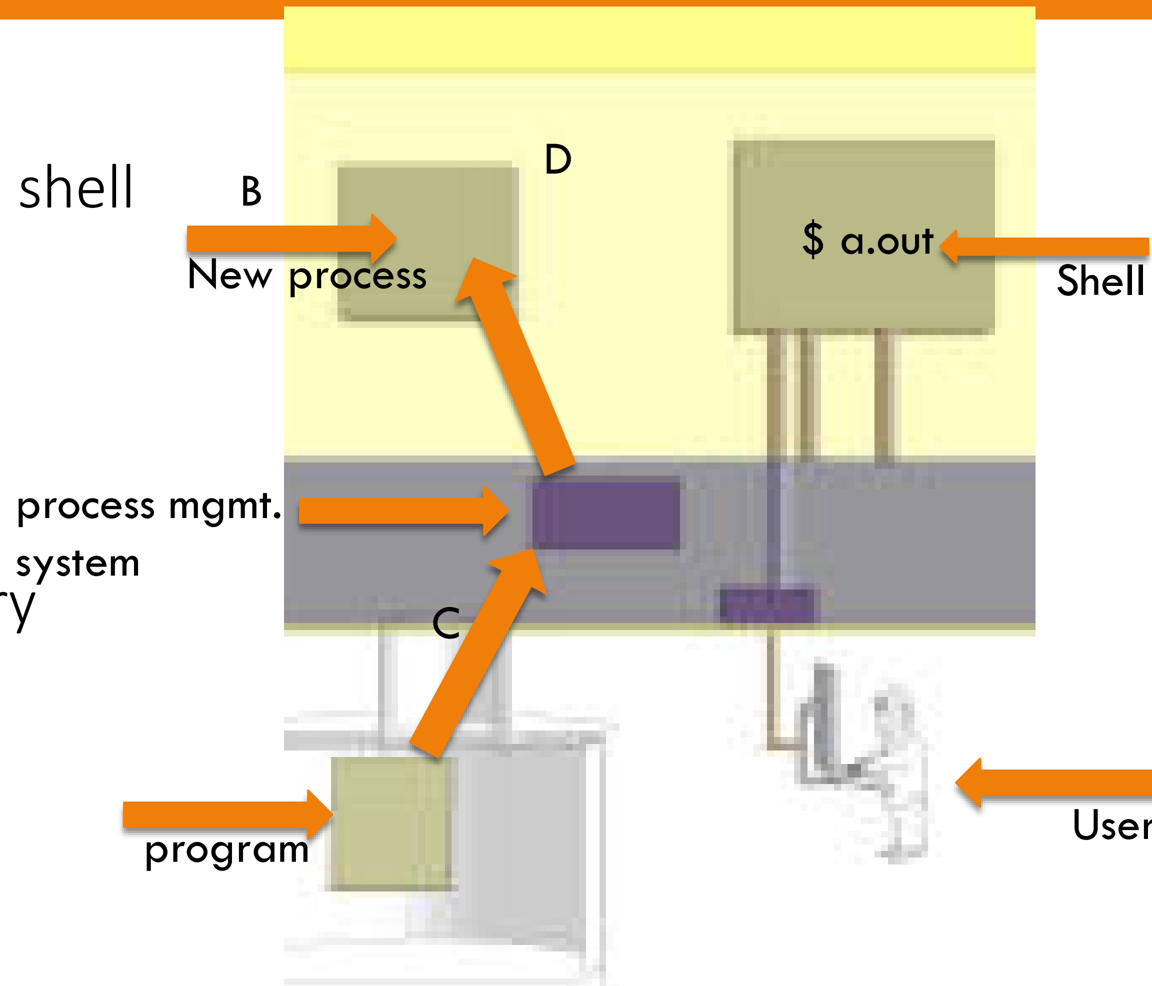
How does the Shell Run Programs?

A. The user types a.out in the shell

B. The shell creates a new process to run the program

C. The shell load the program from the disk into the memory

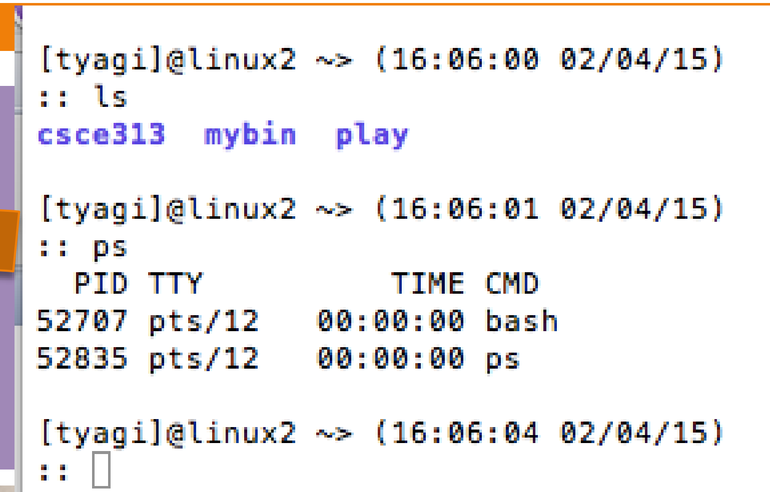
D. The program runs in its process until it is done



Ref: Understanding Unix/Linux Programming by Bruce Molay

The Main Loop of a Shell

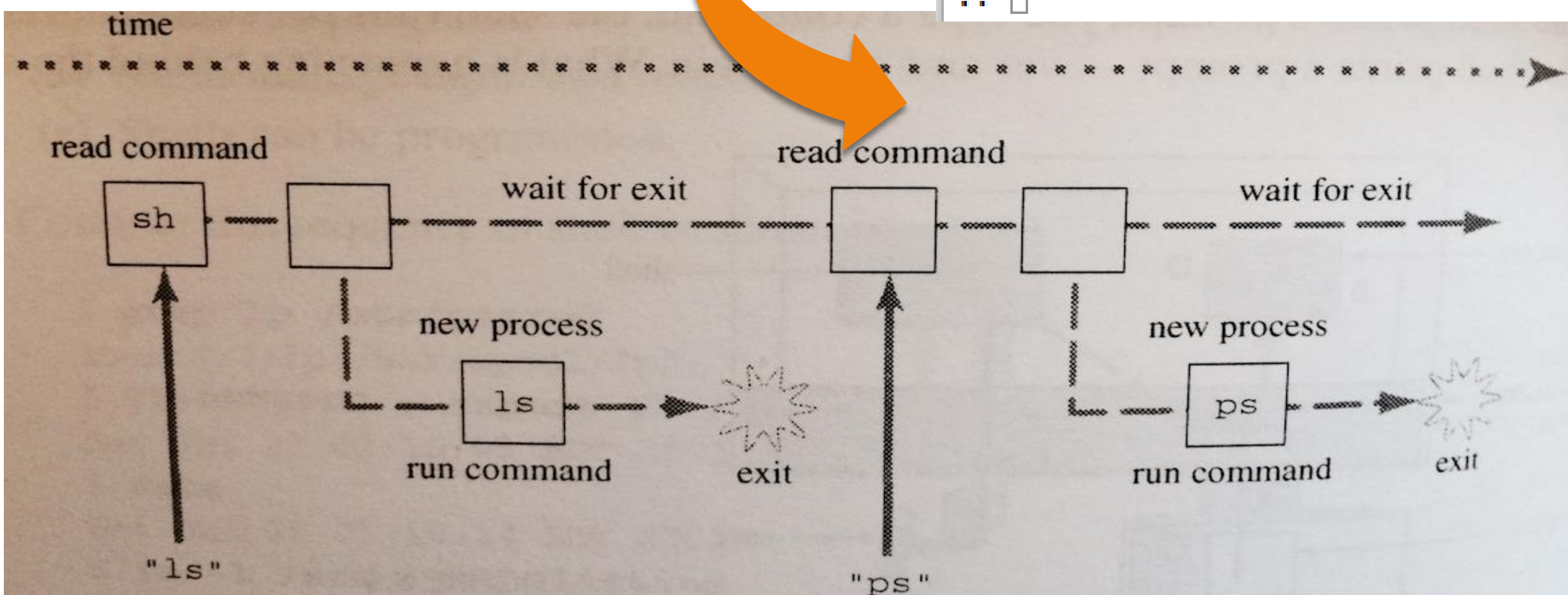
- The shell consists of the following loop:
  - while (! end_of_input)
    - get command
    - execute command
    - wait for command to finish



```
[tyagi]@linux2 ~> (16:06:00 02/04/15)
:: ls
csce313   mybin   play

[tyagi]@linux2 ~> (16:06:01 02/04/15)
:: ps
  PID TTY          TIME CMD
52707 pts/12   00:00:00 bash
52835 pts/12   00:00:00 ps

[tyagi]@linux2 ~> (16:06:04 02/04/15)
::
```



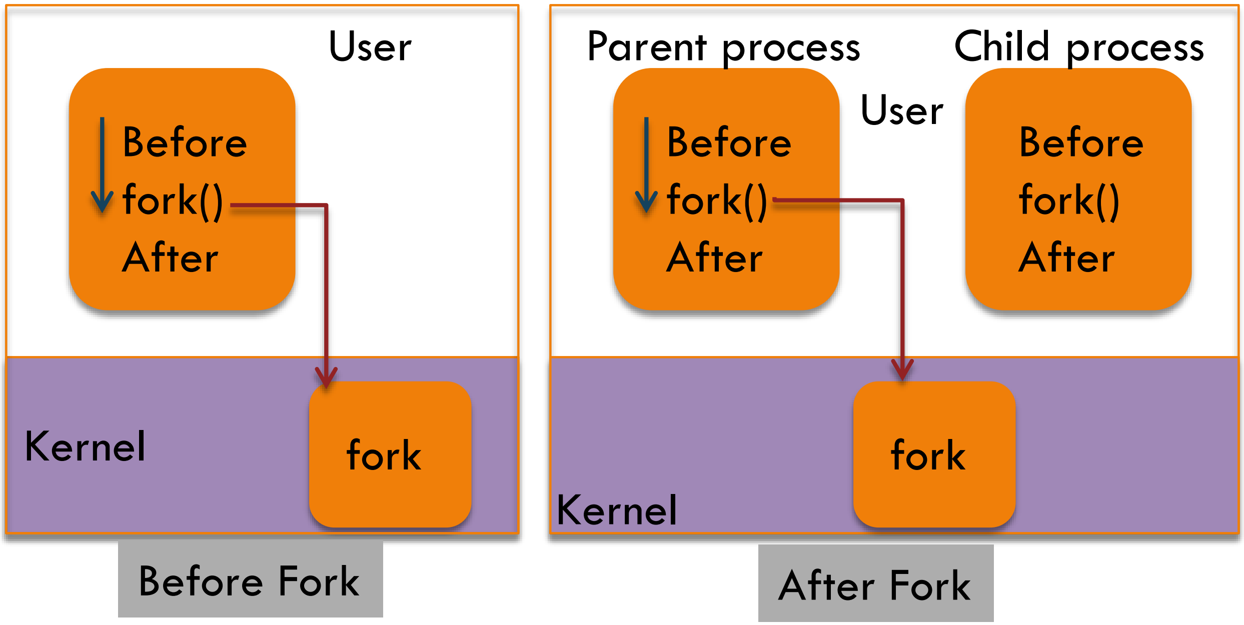
Ref: Understanding Unix/Linux Programming by Bruce Molay

To Write a Shell, we need to...

- Run a Program
- Create a Process
- Wait for Exit

How do we get a new process?

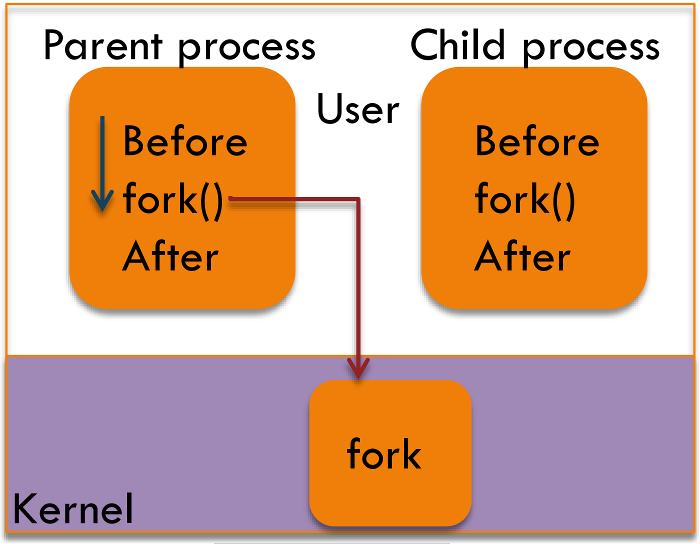
- A process calls FORK to replicate itself
- Usage: fork (); /* takes no arguments*/



- After a process invokes fork, control passes to the KERNEL. The Kernel does this:
  - Allocates address space and data structures
  - Copies the original process into the new process
  - Adds the new process to the set of running processes
  - Returns control back to both processes

How do we get a new process?

- A process calls FORK to replicate itself
- Usage: fork (); /* takes no arguments*/



After Fork

- After a process invokes fork, control passes to the KERNEL. The Kernel does this:
  - Allocates address space and data structures
  - Copies the original process into the new process
  - Adds the new process to the set of running processes
  - Returns control back to both processes

Example: Fork


```
/*  forkdemo1.c
 *shows how fork creates two processes, distinguishable
 *by the different return values from fork()
 */
/* Bruce Molay */

#include<stdio.h>

main()
{
  int ret_from_fork, mypid;

  mypid = getpid();    /* who am i?  */
  printf("Before: my pid is %d\n", mypid);   /* tell the world*/

  ret_from_fork = fork();

  /*  sleep(1);*/
  printf("After: my pid is %d, fork() said %d\n",
         getpid(), ret_from_fork);
}
```

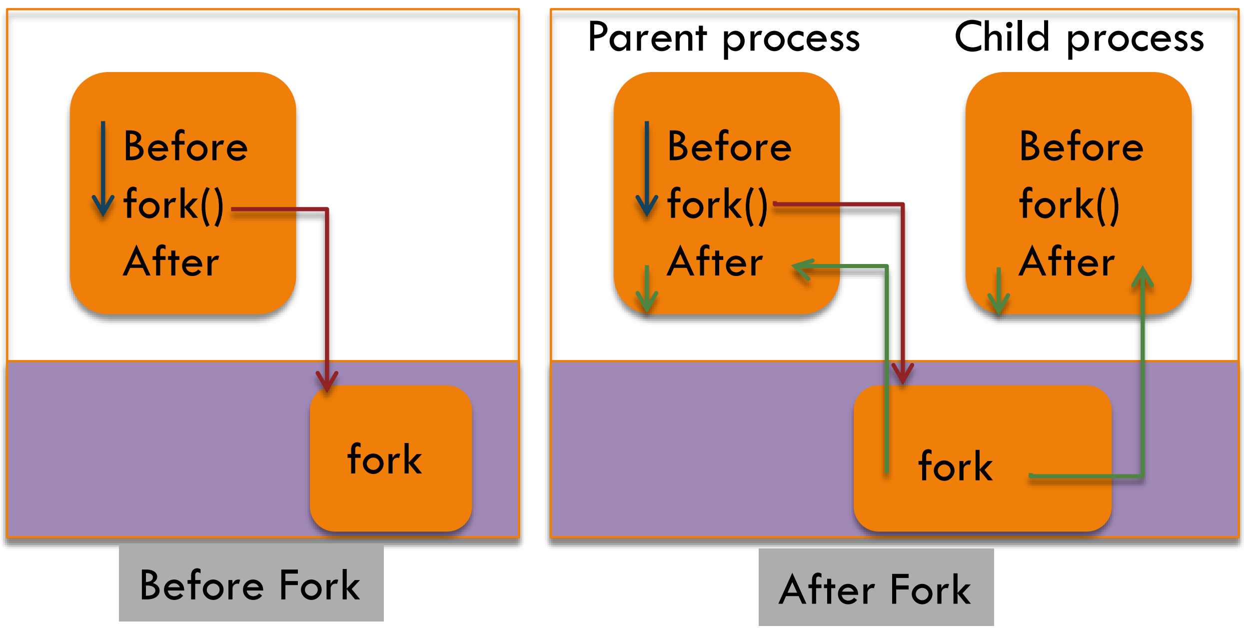
Example: Fork


```
[tyagi]@linux2 ~/csce313/sp15/forkdemo1> (10:38:51 02/09/16)
:: a.out
Before: my pid is 32759
After: my pid is 32759, fork() said 32760
After: my pid is 32760, fork() said 0
```

- Why is the “After” message printed twice but “Before” message only once?
  - Because Fork created a child process and both parent and child execute the rest of the code following the fork

Example: Fork

- Why is the "After" message printed twice but "Before" message only once?

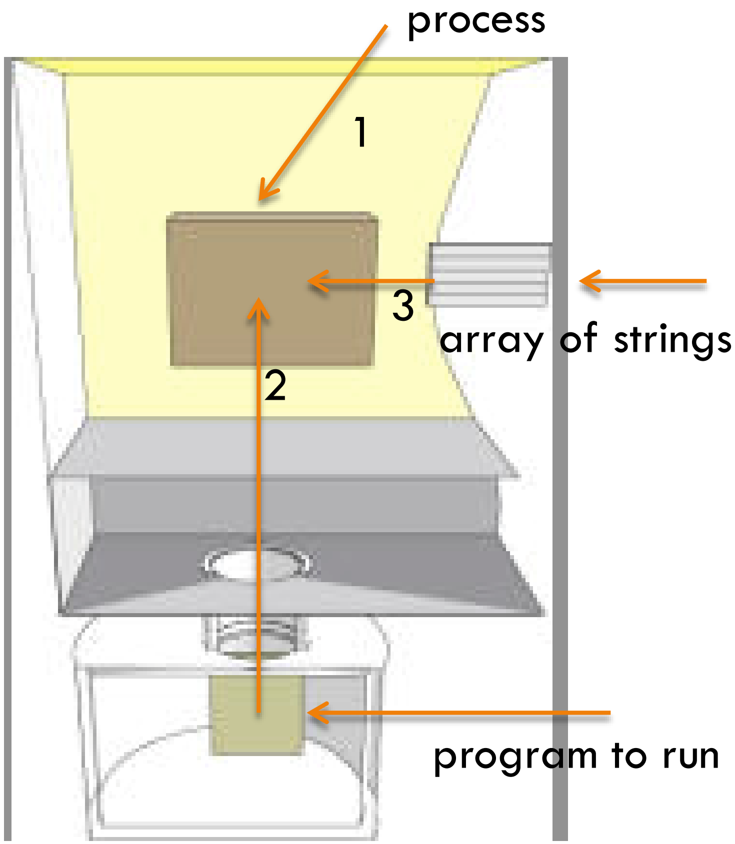


An Observation and A Question

- Observation
  - Fork does a wonderful job of creating a copy of the process that goes on to execute the same code as the parent
- Question
  - If that is the case, how in the world do we get a process to create a child process that does something different than the parent?

How does a Program run a Program?

- Process (Program) calls "execvp"
- Kernel loads program from disk into the process
- Kernel copies arglist into the process
- Kernel calls main(argc, argv)



Example: Program running a program


```
#include <stdio.h>

/* exec1.c - Show how a program runs a program
 */

main()
{
  char*arglist[2];

  arglist[0] = "ls";
  arglist[1] = "-l";
  printf("* * About to exec ls -l\n");
  execvp( "ls" , arglist );
  printf("* * ls is done. bye\n");
}
```

Example: contd.


```
[tyagi]@linux2 ~/csce313/sp15/exec> (10:19:33 02/09/16)
:: ls
exec1.c  expt1.c

[tyagi]@linux2 ~/csce313/sp15/exec> (10:19:34 02/09/16)
:: gcc exec1.c

[tyagi]@linux2 ~/csce313/sp15/exec> (10:19:40 02/09/16)
:: a.out
* * About to exec ls -l
total 3
-rwxr-xr-x 1 tyagi CSE_csfac 11987 Feb  9 10:19 a.out
-rw-r--r-- 1 tyagi CSE_csfac   247 Feb  9 10:18 exec1.c
-rw-r--r-- 1 tyagi games       288 Feb 19  2015 expt1.c
```

Example: contd.


```c
#include <stdio.h>

/* exec1.c - Show how a program runs a program
 */

main()
{
  char*arglist[2];

  arglist[0] = "ls";
  arglist[1] = "-l";
  printf("* * About to exec ls -l\n");
  execvp( "ls" , arglist );
  printf("* * ls is done. bye\n");
}
```

- Where is the second message?
  - *The exec system call clears out the machine language code of the current program from the current process and then in the now empty process puts the code of the program named in the exec call and then runs the new program*
- execvp does not return if it succeeds
- execvp is like a brain transplant

Fork and Exec


```
int main(int argc, char* argv[] ) {
 int pid = fork();
     if ( pid == 0 ) {
     execvp("date", argv);
}
/* parent sleeps for 2sec to let child go first*/
  wait( 2 );
  printf( "Finished the parent process\n"
        " - the child won't get here--you will only
see this once\n" );
  return 0;
}
```

```
:: ./a.out
Sun Jan 29 14:43:45 CST 2017
Finished the parent process
 - the child won't get here--you will only see this once
```

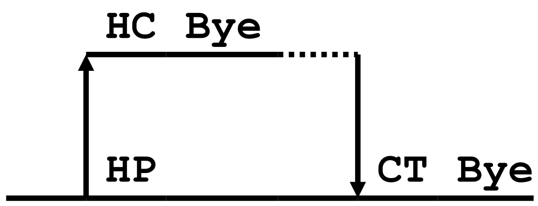
`wait`: Synchronizing With Children

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is `pid` of child process that terminated
  - If `child_status != NULL`, then integer it points to will be set to indicate why child terminated

wait: Synchronizing With Children


```
Void wait_demo() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit(0);
}
```



Some questions to ponder about processes

- ✓ How is a process created?
- □ How is a process deleted?
- □ Is there a user process and kernel process
- □ Where do we keep information about a process
- ✓ Does a process have to run through completion from start to finish or can it be interrupted?
- □ Do processes have priorities?
- □ What are the relationships between multiple processes in a system?
- □ Can we have multiple processes related to the same program? Would multiple processes of the same program share addresses during execution?
- ✓ How does a program run a program?
- ✓ How does a parent wait for a child to exit?

Key Learnings

- Shell Basics
- Replacing Program Executed by Process
  - Call `execv` (or variant)
    - One call, (normally) no return
- Spawning Processes
  - Call to `fork`
    - One call, two returns
- Reaping Processes
  - Call `wait`

What's coming up in Week 4?

- More about process fork, exec, and new functions related to process data and control
- Process Life-Cycle
- What does it take to execute the life-cycle?
- Orphan, Zombie Processes
- Problem Solving related to Process Execution