Systems I – CS 6013

Computer Architecture and Operating Systems

Lecture 7: Processes 1

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*Adapted from Ryan Stutsman's slides

Miscellaneous

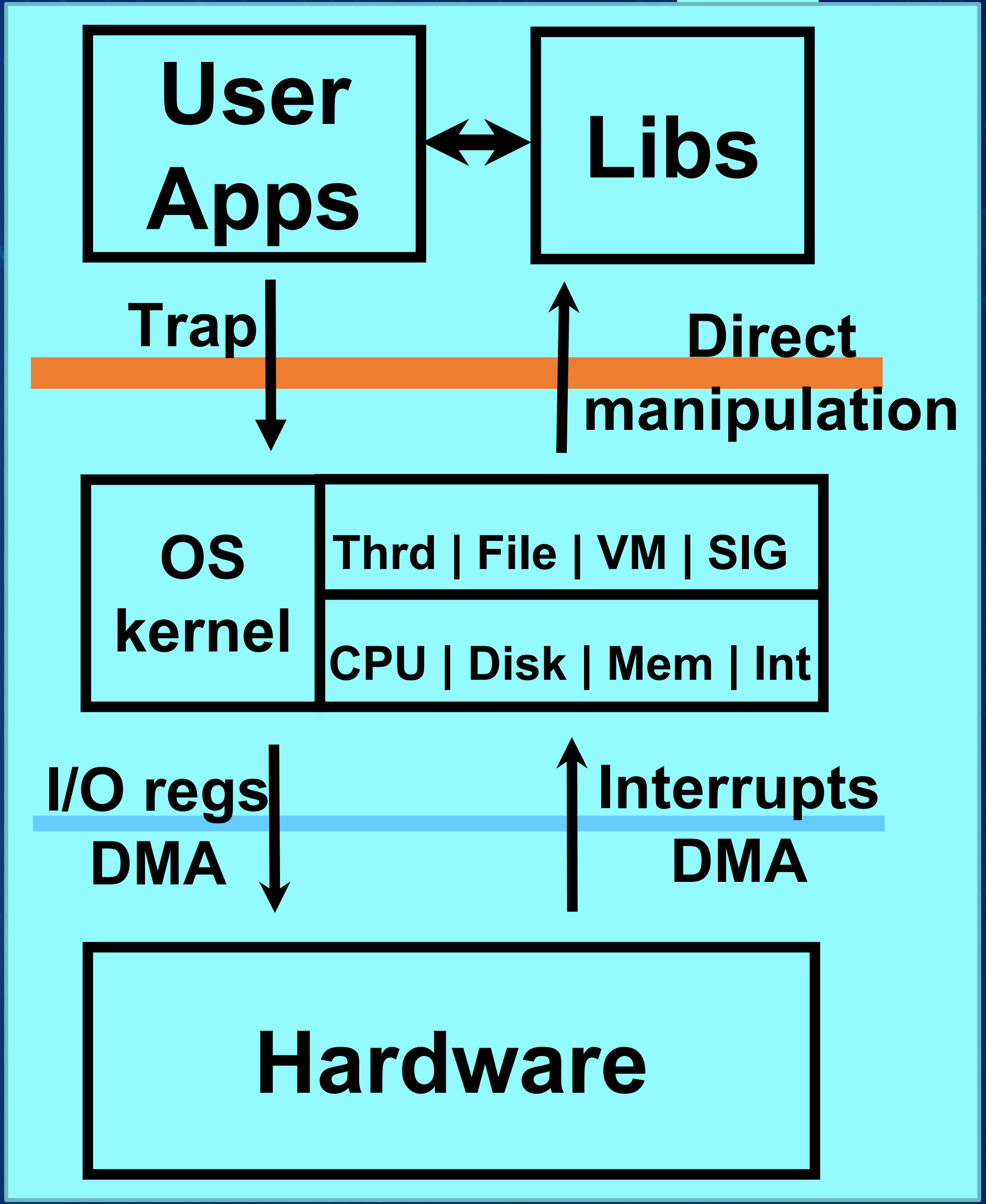
- Dereferencing a Pointer
  - Address vs Value
- Setting up and taking down new Stack Frame – epilogue / prologue

Lecture 6 – Topics

- Processes
  - Memory Layout
  - PCB – Process Control Block
  - Creating Processes – Fork()

Standard OS Structure

- User-level
  - Applications
  - Libraries: many common “OS” functions
  - Example: malloc() vs sbrk()
- Kernel-level
  - Top-level: machine independent
  - Bottom-level: machine dependent
  - Runs in protected mode
  - Need a way to switch (user <-> kernel)
- Hardware-level
  - Device maker exports known interface
  - Device driver initiates operations by writing to device registers
  - Devices use interrupts to signal completion
  - DMA – Direct Memory Access (offloads work, but has restrictions)

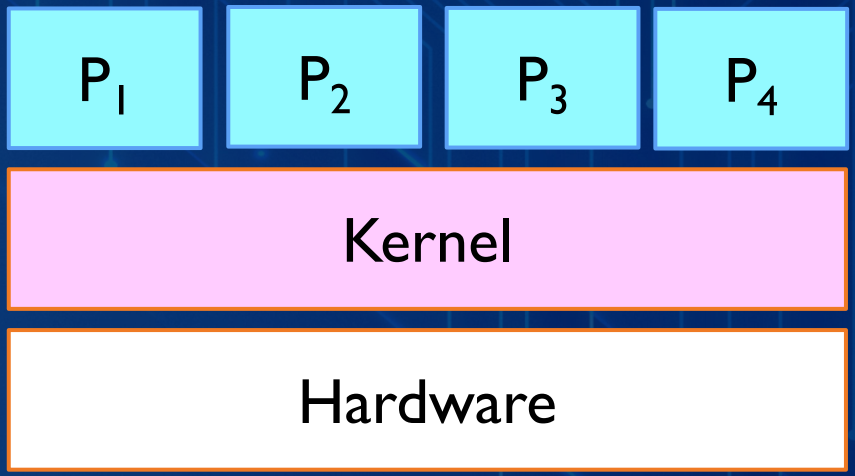


What is a Process?

- **Process**: execution context of running program
- A **process** does not equal a **program**!
  - Process is an *instance* of a program. Where have we talked about instances before?
  - Many copies of same program can be running at same time
- OS manages a variety of activities
  - User programs
  - Batch jobs and scripts (which are just other programs)
  - System programs – print spool, file servers, net daemons
    - BTW, what is a daemon? (Pronounced Demon)
      - A process that runs in the background… Usually on a long term basis. (eg: sshd, ntpd, httpd, …)
- Each of these activities is encapsulated in a process
- Everything happens either in kernel or a process
  - The OS is not a process

Isolating Processes

- Lots of running processes
- Each with own code, data
- Each need to interact with devices, memory, CPU
- How do we **multiplex** the hardware among them?
- How do we make this safe? Efficient?

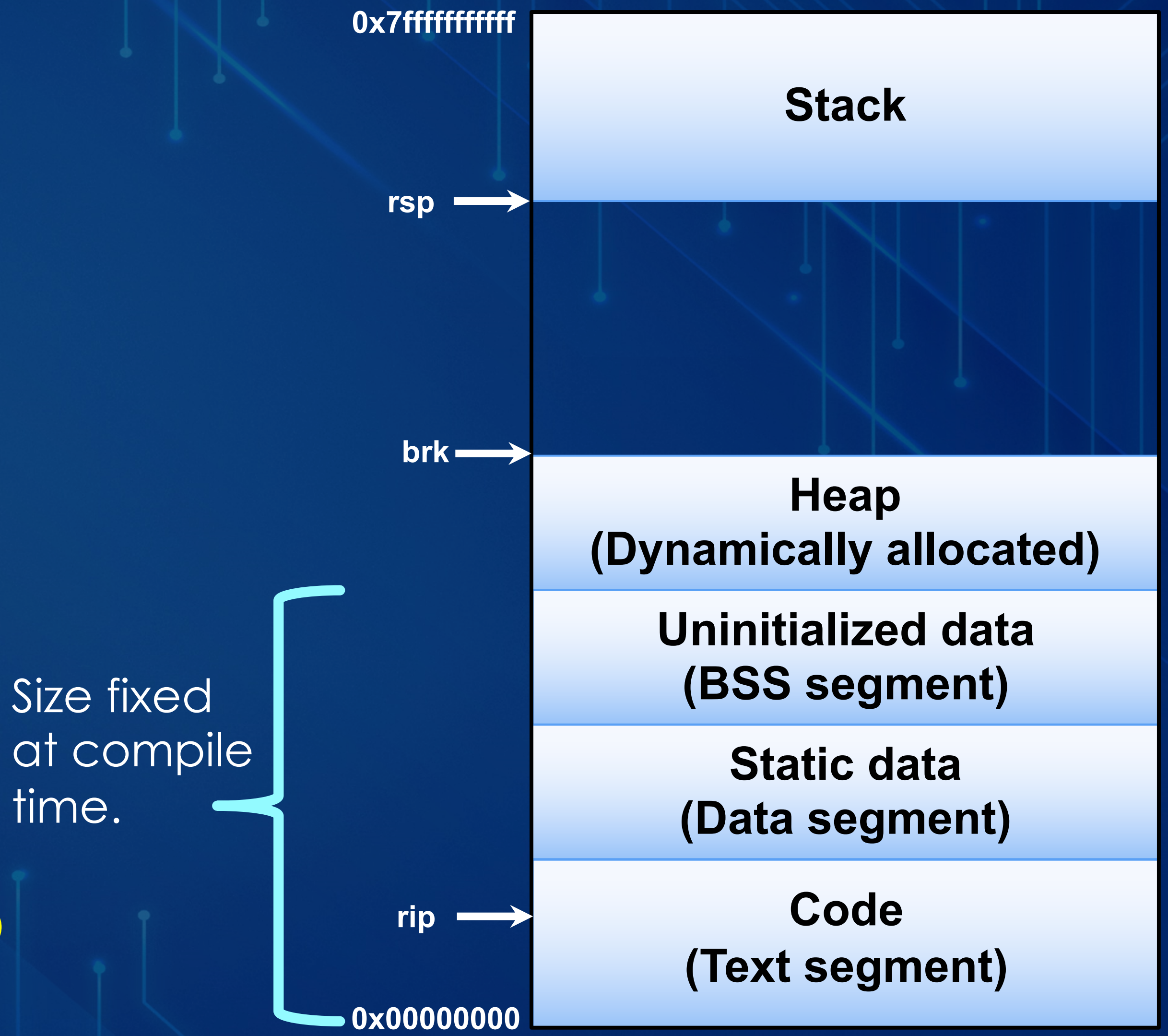


What is in a Process?

- Process state consists of:
  - Memory state: code, data, heap, stack
  - Processor state: IP, registers, etc.
  - Kernel state:
    - Process state: ready, running, etc.
    - Resources: open files/sockets, etc.
    - Scheduling: priority, CPU time, etc.
- **Address space** consists of:
  - Code
  - Static data (data and BSS)
  - Dynamic data (heap and stack)
  - See: Unix `size` command
- Special pointers:
  - IP: current instruction being executed
  - brk: top of heap (explicitly moved)
  - sp: bottom of stack (implicitly moved)

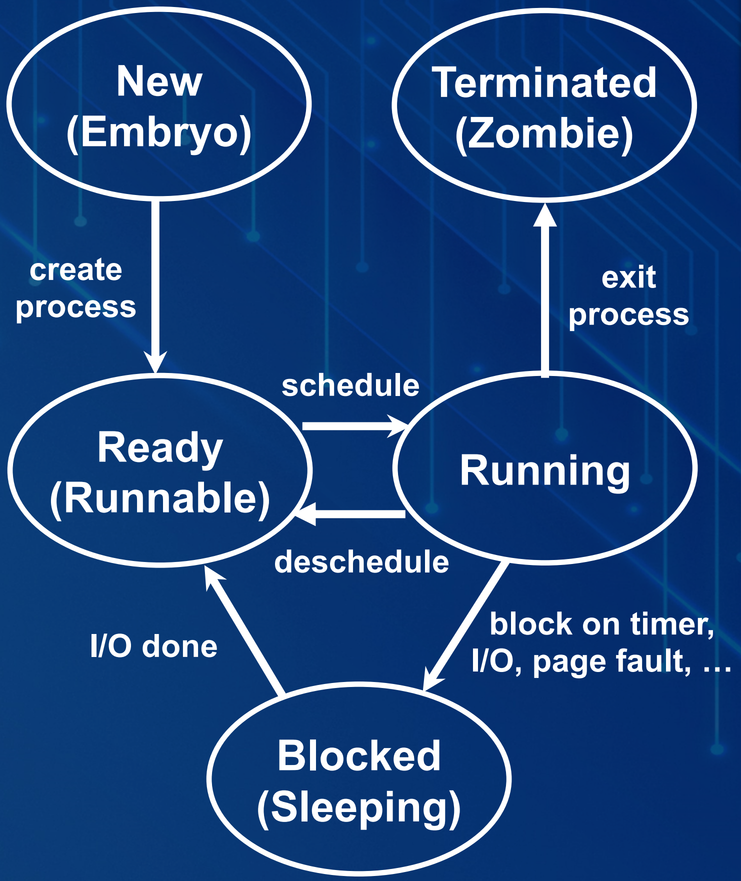
All tracked in a **Process Control Block (PCB)**

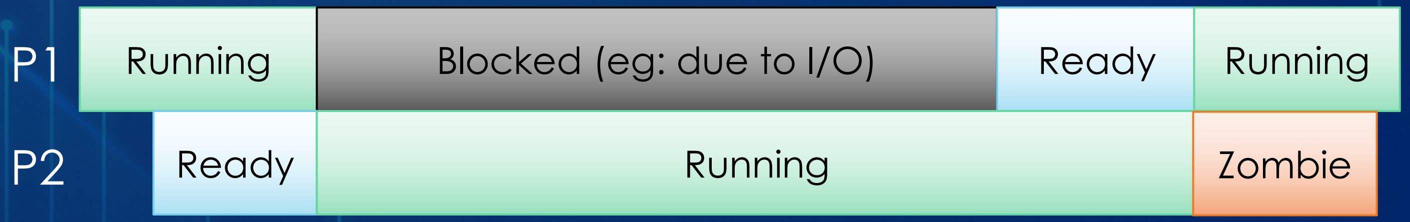
Process Memory Layout



Process State Machine

- Each process has a state:
  - new: OS is setting up process
  - ready: runnable, but not running
  - running: executing instructions on CPU
  - blocked: stalled for some event (e.g., IO)
  - terminated: process is dead or dying
- Process moves from state to state as a result of its actions or external actions
  - Program: sleep(), IO request, …
  - OS action: scheduling
  - External: interrupts, IO completion
- Key to efficiency and high utilization
  - Blocked processes yield CPU
  - They are just data structures



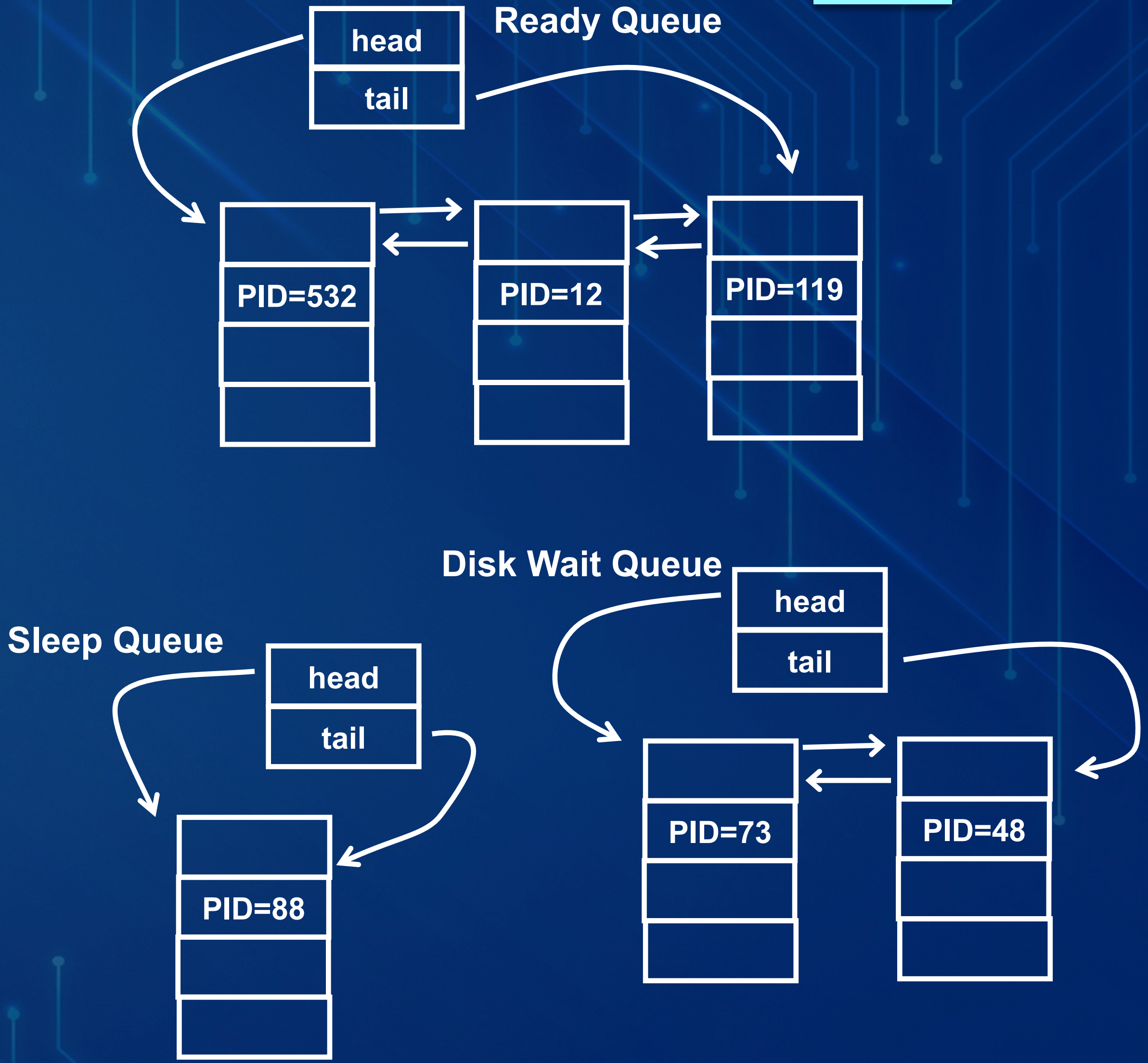


Process Control Block (PCB)

- the PCB is a data structure - One per process, allocated in kernel memory
- Stores state of a process, typically including:
  - Process state (running, waiting, …)
  - PID (process identifier, often a 16-bit integer)
  - Machine state: IP, SP, registers (if not currently running)
  - Memory management info
  - Open file table (open socket table)
  - Queue pointers (waiting queue, I/O, sibling list, parent, …)
  - Scheduling info (e.g., priority, time used so far, …)
- On process creation, the kernel will: allocate PCB, initialize, put on ready queue (queue of runnable processes)
- On exit: clean up all process state (close files, release memory, page tables, etc)

Process State Queues

- OS tracks PCBs using queues
- Ready processes on the "ready queue"
- Each I/O device has a wait queue
    - Queue traversed when I/O interrupt handled
- OS invariant: A process is always running. Other processes are either on the ready queue, or on a single wait queue
    - Implications of this?
- Processes linked to parents and siblings
    - Needed to support wait()



xv6 PCB

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- What does "e" (in registers listed) tell you about this system?
  - 32-bit computer

```
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                      // Proc mem size (bytes)
  pde_t* pgdir;                 // Page table
  char *kstack;                 // Bottom of kstack
  enum procstate state;         // Process state
  int pid;                      // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frm for syscall
  struct context *context;      // swtch() here to run
  void *chan;                   // If !0, sleeping on chan
  int killed;                   // If !0, have been killed
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name
};
```

Process Management

- OS manages processes:
  - Creates, deletes, suspends, and resumes processes
  - Schedules processes to manage CPU allocation
  - Manages inter-process communication and synchronization
  - Allocates resources to processes (and takes them away)
- Processes use OS functionality to cooperate
  - Signals, sockets, pipes, files to communicate

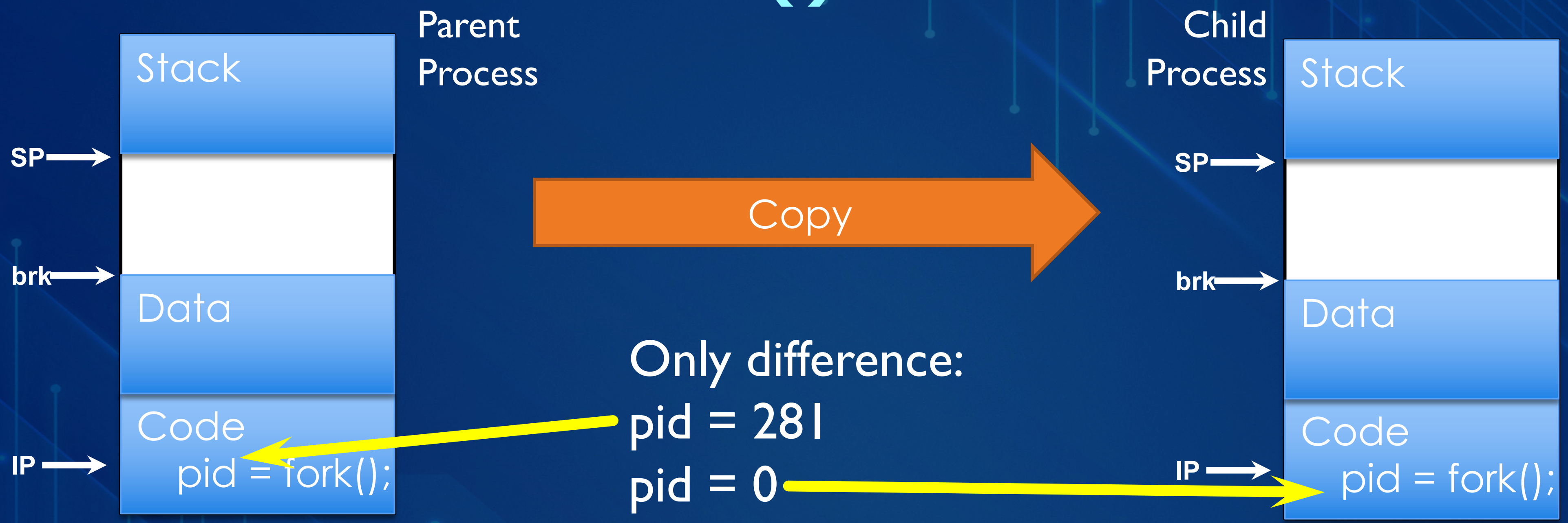
Practical Process Management

- On a Unix machine, try:
  - ps –Af # lots of info on all running processes
  - kill –9 547 # terminates process with PID 547 (Don't actually try this one…?!?)
  - top # displays dynamic info on top running jobs
  - Write a program that calls:
    - getpid(): returns current process's PID (process id)
    - fork(): create a new process
    - wait(): wait for exit of a child process
    - exec(): load a new program into the current process
    - sleep(): puts current process to sleep for specified time
- Commands work on macOS / linux
- On Windows → Task manager (CTL-ALT-DEL)

Creating Processes: `fork()`

- Creates process that is near-clone of forking parent
  - Address space and running state is cloned
- Return of `fork()` differs:
  - 0 in child
  - child PID in parent
  - 3rd Option?
    - -1 – ERROR! Check error codes.
- Many kernel resources are shared between the parent and child…
  - open files and sockets
  - have to be careful
- `wait()` lets a process wait for the exit of a child
- To spawn a new program, use some form of `exec()`

Semantics of fork()



- fork(), exit(), and exec() are weird!
  - fork() returns twice – once in each process
  - exit() does not return at all
  - exec() usually “does not return”: replaces process’ program

fork() and wait() demo

- Note: the `fork()` call can return first in child or parent process…

Fork() / Wait ()


```
int main(){
    pid_t pid = fork();
    if( pid < 0 ) {
        perror( "Fork failed…" );
        return -1;
    }
    else if( pid == 0 ) {
        printf( "HC: hello from child\n" );
        exit(17);
    }
    else {
        int child_status;
        printf( "HP: hello from parent\n" );
        waitpid( pid, &child_status, 0 ); // Waits for child to end
        printf( "CT: child result %d\n", WEXITSTATUS( child_status ) );
    }
    printf( "Bye\n" );
    return 0;
}
```

~ Fin ~