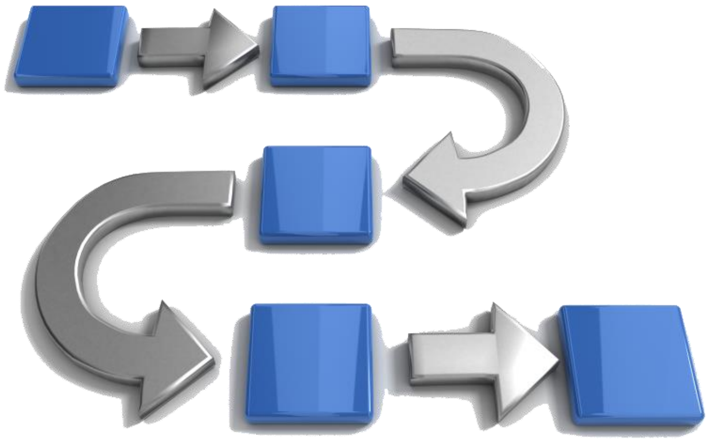


Processes

Managing User Software

Modern Operating Systems, by Andrew Tanenbaum Chap 2

Operating Systems: Three Easy Pieces (a.k.a. the OSTEP book) Chap 4

man pages in any UNIX system

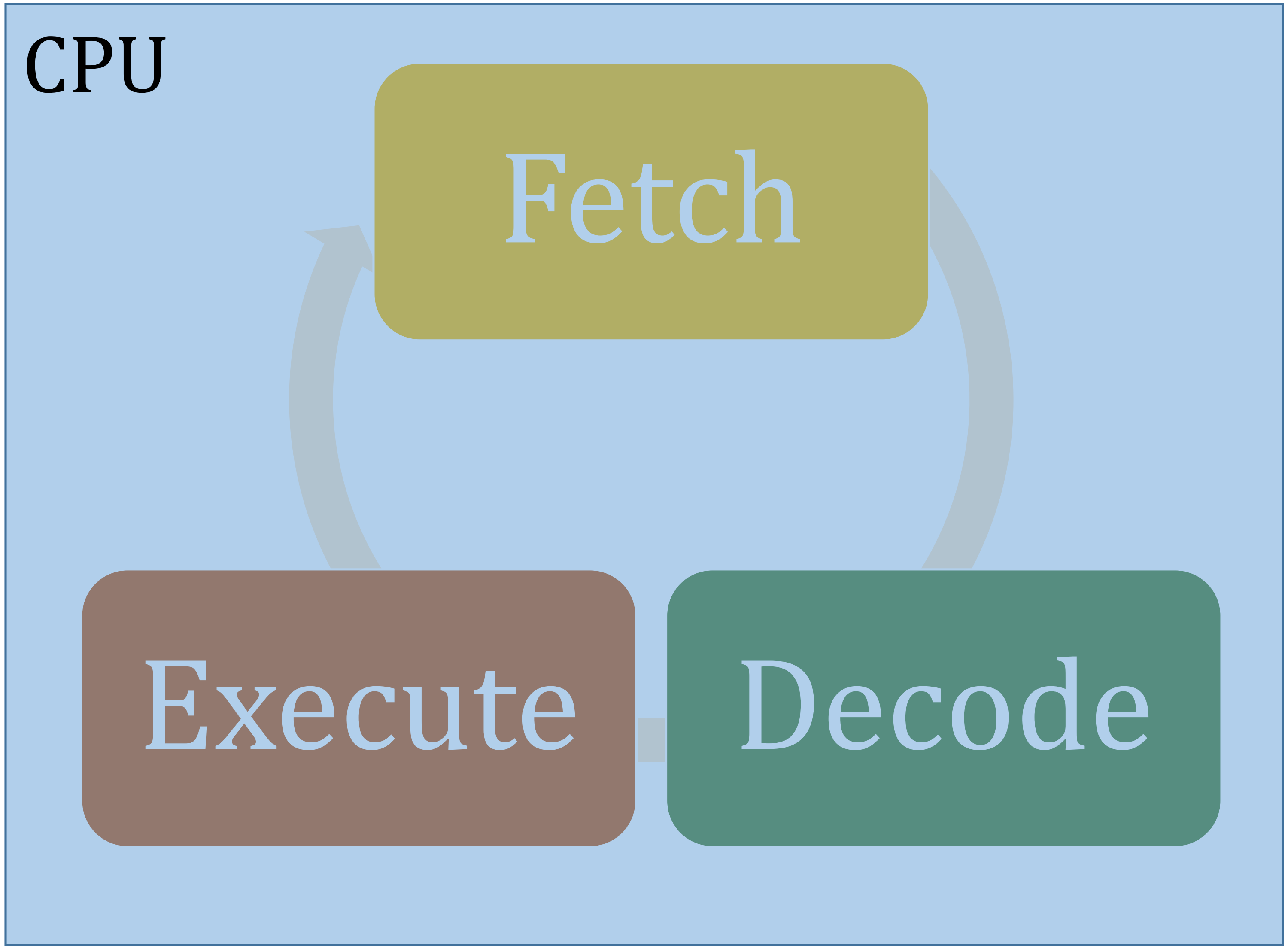
Process Terminology

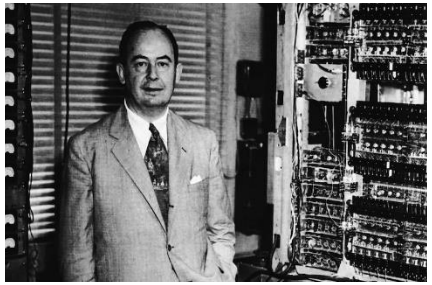
- A *program* is a set of instructions somewhere (like in a file)

- A *process* is the execution of a program
  - And all the resources required to execute that program

- When execution starts, the OS reserves memory for the process, and loads the program into that memory
  - Then sends the CPU to execute the first instruction of the program

Von Neumann model of Computing





Process vs. Program

Program

- Passive entity
  - Static code and data
- Stored on disk

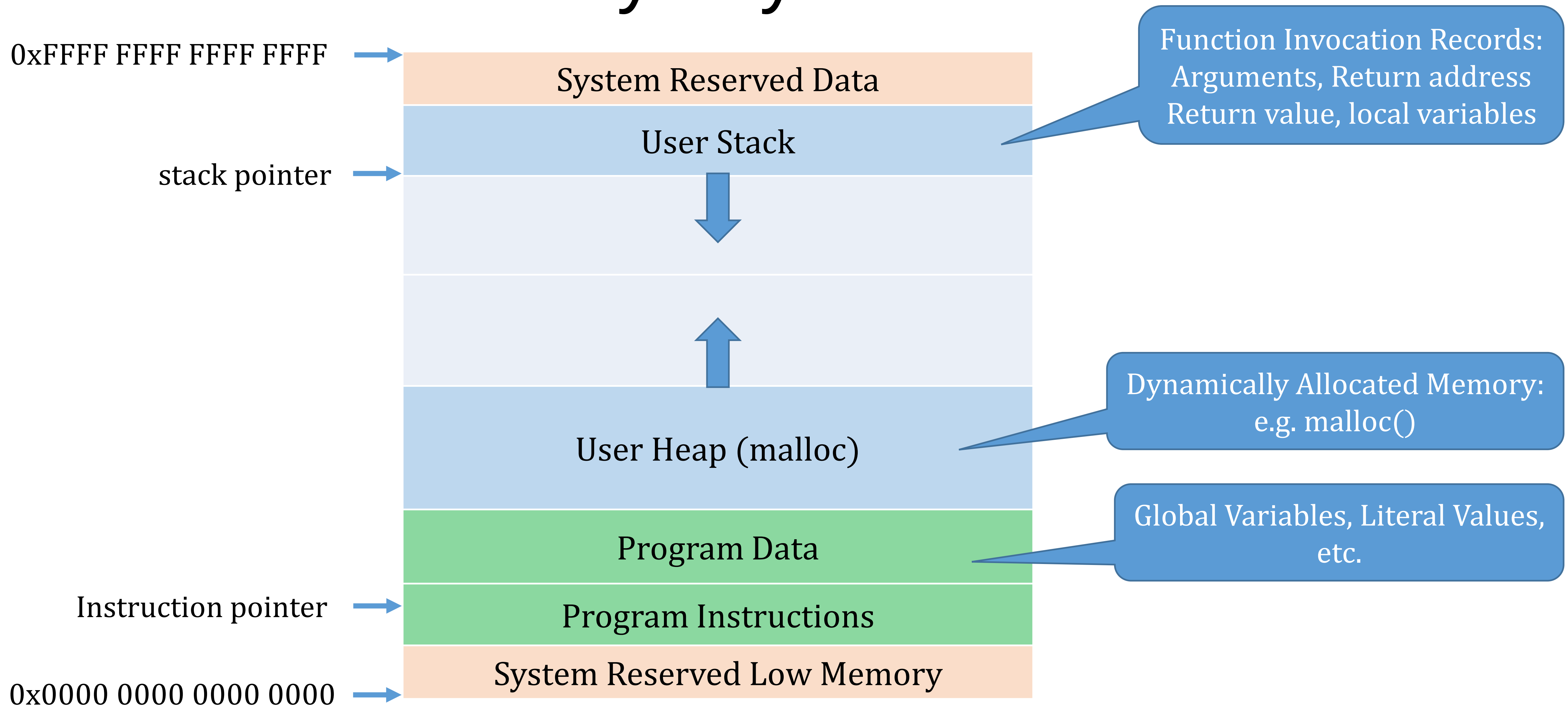
Process

- Program
  - actively executing code
  - Static code and data
  - Dynamic code and data
- Memory
- Dynamic Data (e.g. registers)
- Many processes can run the same program (e.g. "ls")

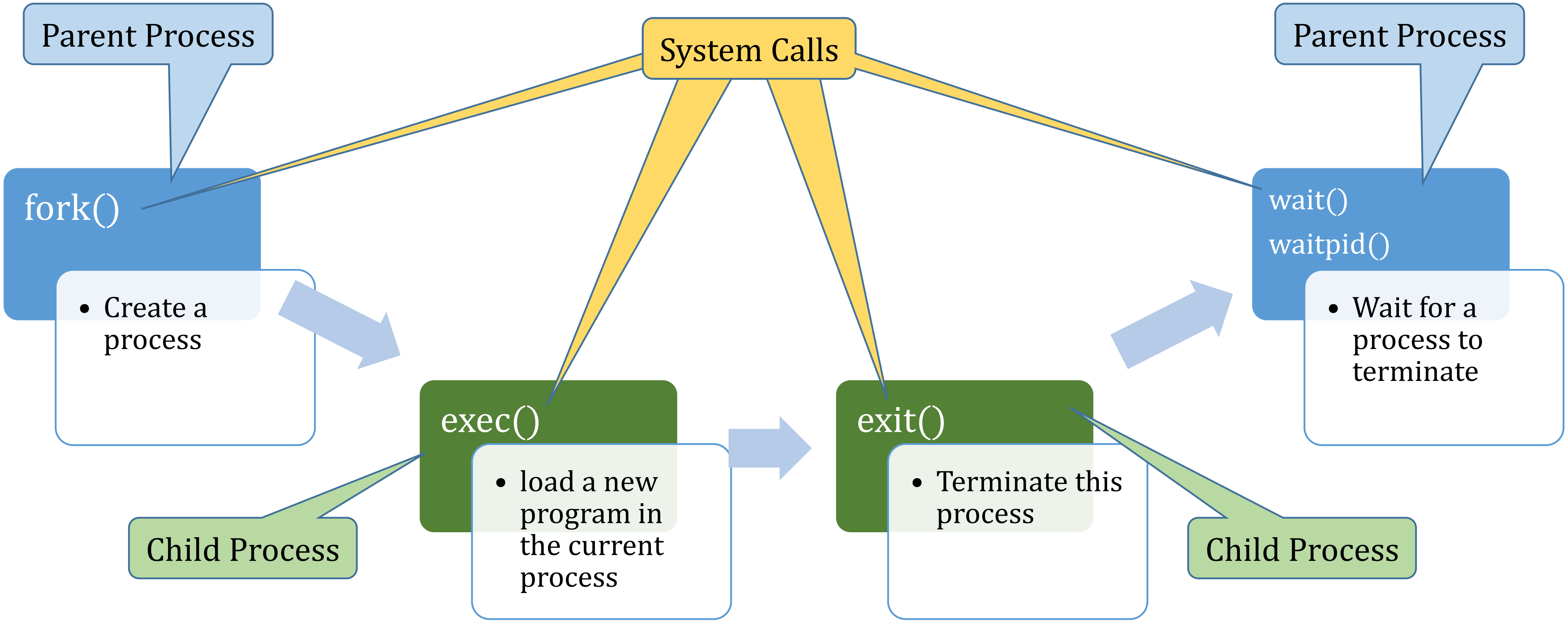
What's in a process?

- Memory space (static and dynamic)
- Procedure call stack
- Registers and counters
  - Program counter, stack pointer, general purpose registers
- Open files and connections
- And more.

Process Memory Layout



Process Life Cycle



Process Creation (Using "fork")

- "root" OS process started at bootstrap time
- "root" process forks out service daemons
  - See /etc/init.d on Linux for scripts that start services
- Login forks terminal window shell process
- Shell forks new process when you run a command
- Some user programs invoke fork

Example Process Creation

See: Examples/process/fork_ex.c

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

int main() {
   pid_t pid; int status; int ret;
   pid = fork();
   if (pid < 0) {
      perror("fork failed:");
      exit(1);
   }
```

```
   if (pid == 0) { // Child executes this block
      printf("This is the child\n");
      exit(99);
   }
   if (pid > 0) { //Parent executes this block
      printf("This is parent. The child is %d\n", pid);
      ret = waitpid(pid, &status, 0);
      if (ret < 0) {
         perror("waitpid failed:");
         exit(2);
      }
      printf("Child exited with status %d\n", WEXITSTATUS(status));
   }
   return 0;
}
```

The strange behavior of fork

- The fork() function is called once… but it returns TWICE!!!
  - Once in the parent, once in the child
  - parent and child are two different processes

- The child is an exact copy of the parent
  - Address space is copied – that means the program is copied as well!
  - Registers and pointers are also copied – e.g. instruction pointer
  - Only difference is the return value… in child=0, in parent=<child PID>
    - Process ID (PID) – a unique 4 digit number assigned to the process

After a fork()

- Now there are two processes running the same program
- That program can use the return code to make the child and parent do different things
- But we really want the child to run a DIFFERENT program

- We can do that with the exec() system call

Example of Exec

See: Examples/process/exec_ex.c

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

int main() {
  pid_t pid;
  if ((pid = fork()) < 0) {
    fprintf(stderr,
      "fork failed\n");
    exit(1);
  }
```

```
  if (pid == 0) {
    if(
execlp("echo","echo","Hello from the child",(char *)NULL)
             == -1) {
      fprintf(stderr, "execlp failed\n");
    }
    exit(2);
  }
  printf("parent carries on\n");
  return 0;
}
```

The strange behavior of exec

- exec() and all it's variant forms first checks to see if the arguments are valid. If not, exec issues an error and returns

- If the arguments are OK, exec() :
  - Loads a new program into the address space
  - Resets the stack and heap
  - Maintains all old I/O descriptors and status (Useful for implementing filters)
  - Modifies instruction pointer to start at the beginning of the program
  - Runs the code (fetch/decode/execute) until an exit occurs

- As a result… unless there is an error, exec() DOES NOT RETURN!
  - The old program is gone!

Shell Pseudo-Code

```
while(1) {
    print prompt (>) to terminal
    read command from terminal
    pid=fork()
    if (pid==0) {
        rc=exec(command)
        if (rc!=0) { print error; continue }
    }
    cmdrc=waitpid(pid)
}
```

Q: Does child process ever wait?

Different flavors of exec()

- `int execl(char * pathname, char * arg0, … , (char *)0);`
  - Full pathname + long listing of arguments
- `int execv(char * pathname, char * argv[]);`
  - Full pathname + arguments in an array
- `int execle(char * pathname, char * arg0, … , (char *)0, char envp[]);`
  - Full pathname + long listing of arguments + environment variables
- `int execve(char * pathname, char * argv[], char envp[]);`
  - Full pathname + arguments in an array + environment variables
- `int execlp(char * filename, char * arg0, … , (char *)0);`
  - Short pathname + long listing of arguments
- `int execvp(char * filename, char * argv[]);`
  - Short pathname + arguments in an array

More Info: "man 3 exec"

Terminating a process

- Return from top level function (main)
  - Return value (int) is the exit status of the process

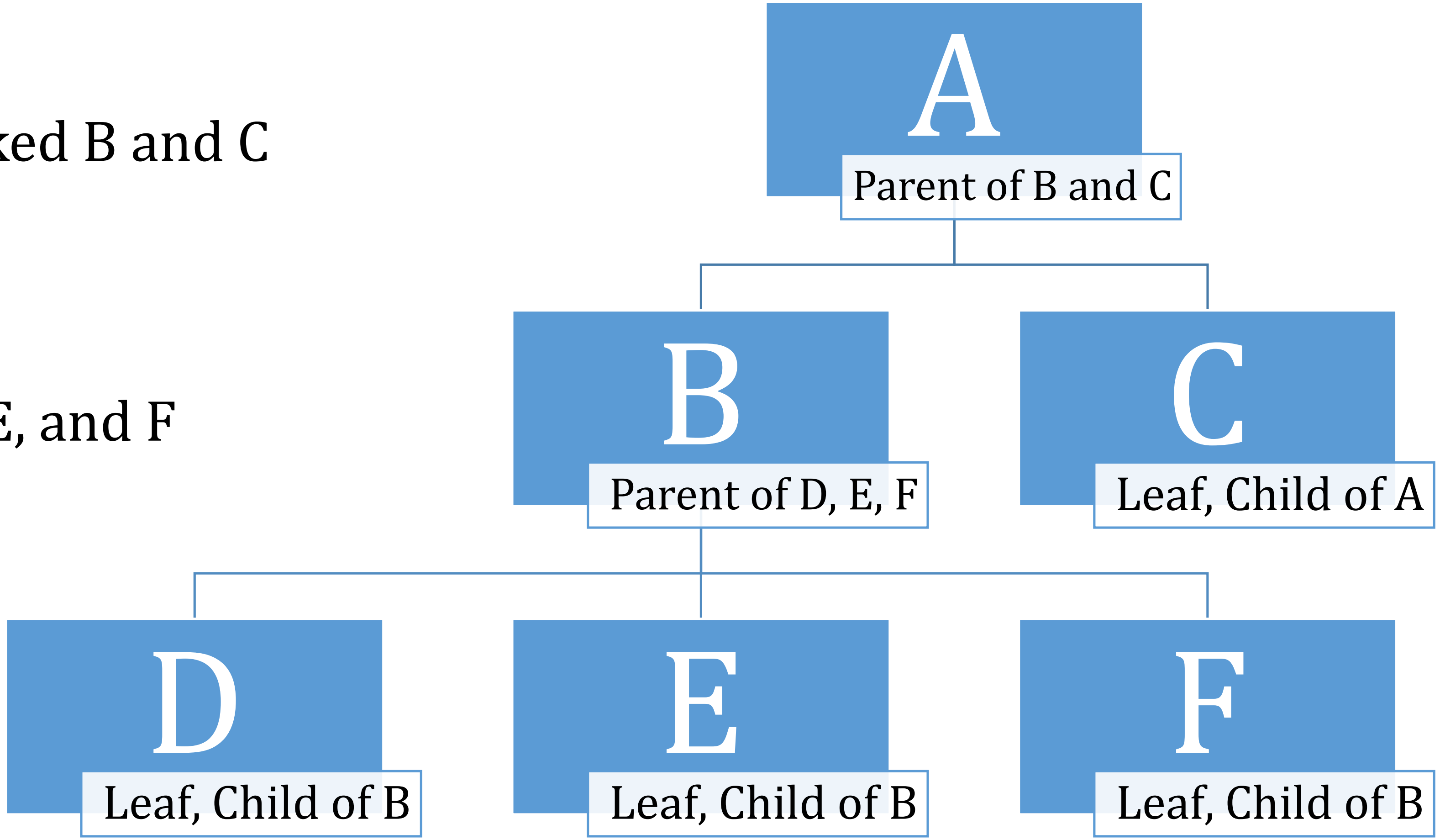
- Invoke the exit(status) library function (in stdlib.h)
  - The argument (int) is the exit status of the process
  - See http://man7.org/linux/man-pages/man3/exit.3.html

- 0 or EXIT_SUCCESS if process worked
- non-zero or EXIT_FAILURE if an error occurred

Process Hierarchy Tree

A forked B and C

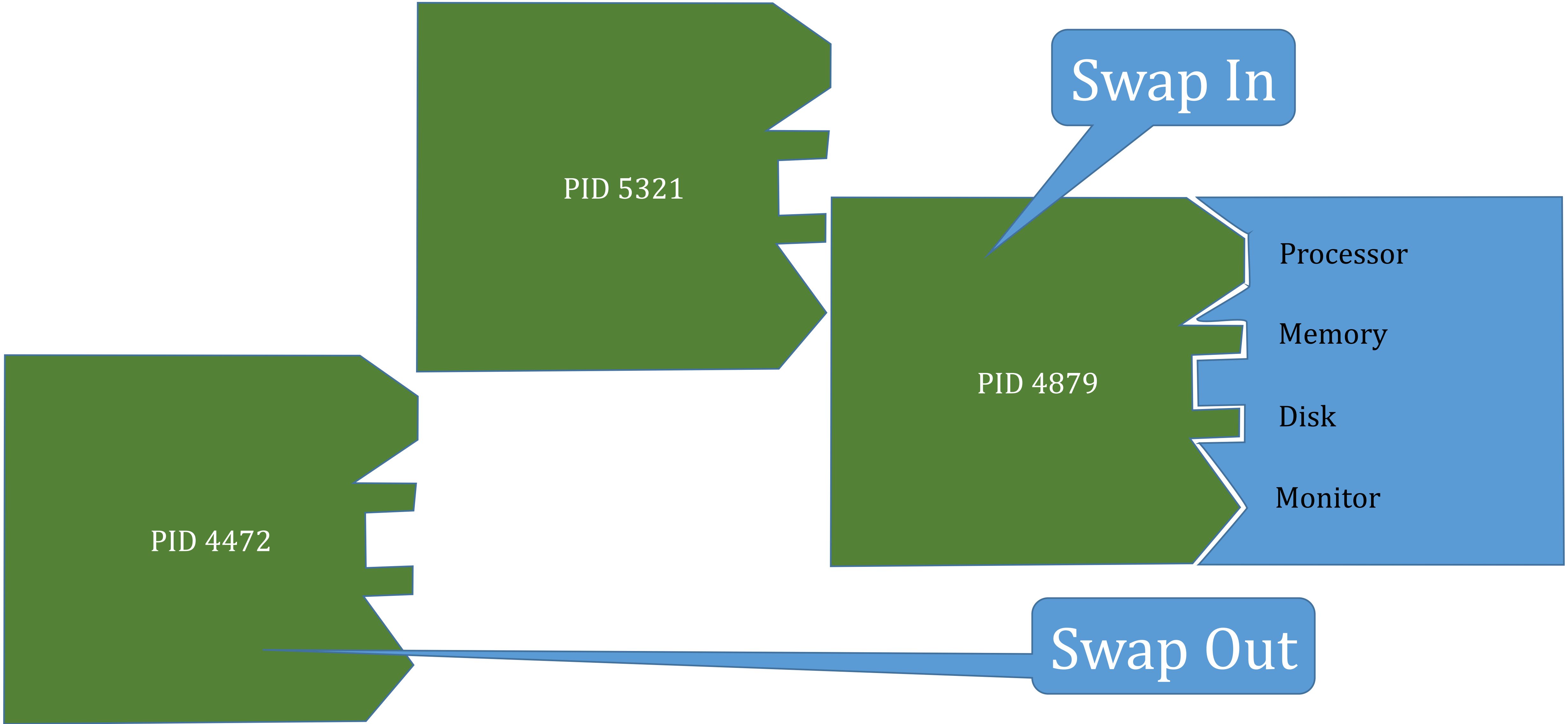
B forked D, E, and F



Process Resources

- Each process THINKS it owns all machine resources
  - “virtual” processor, virtual memory, virtual keyboard, virtual monitor, virtual disks, virtual network, …
- OS connects VIRTUAL resources to REAL resources

Time Slicing



Process Swapping

- Time required to save the swap-out process state
  - Memory, Registers, IO status, etc. etc. etc.
- Time required to restore the swap-in process state
  - Memory, Registers, IO status, etc. etc. etc.
- All this is time when neither process can move forward
  - Overhead

- Problem: Time slice needs to be long enough to minimize swap overhead, but short enough so user's don't perceive swaps

Process Swapping and Virtual Memory

- Virtual Memory: Address space divided into 4K pages
- Only those pages we are actually using are in real memory
  - Most 4K pages are either never referenced and never instantiated
  - Or were referenced a "long" time ago, and are kept on disk

- Real memory holds pages for multiple processes
  - Each page is connected to a specific process

- When real memory is full, the "oldest" page is swapped out
  - No matter what process it's connected to

Process Swapping and Virtual Memory

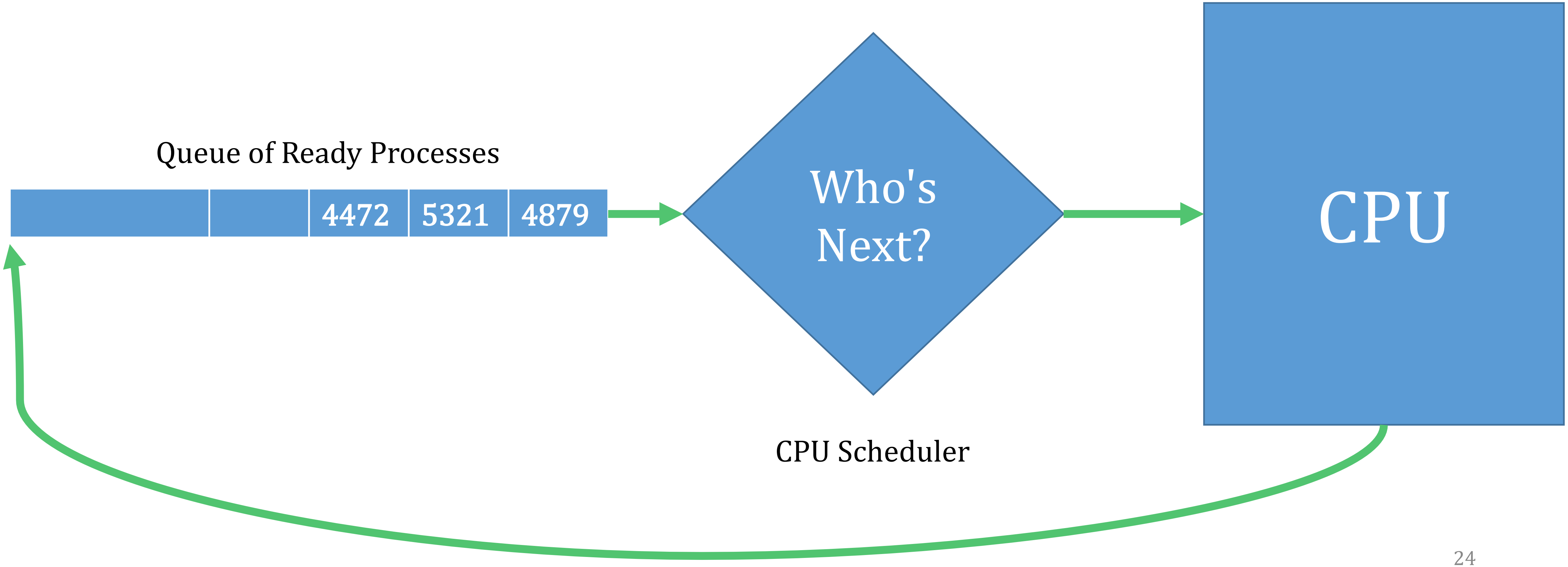
- No need to swap active memory pages out of real memory!
  - The will age, eventually become old enough to get swapped out of real memory OR the process will become active and use those pages again

- When a process gets blocked, it may have many pages active
- When it resumes, at least some of those pages may be swapped out and need to be reloaded
  - But the impact is virtually imperceptible!

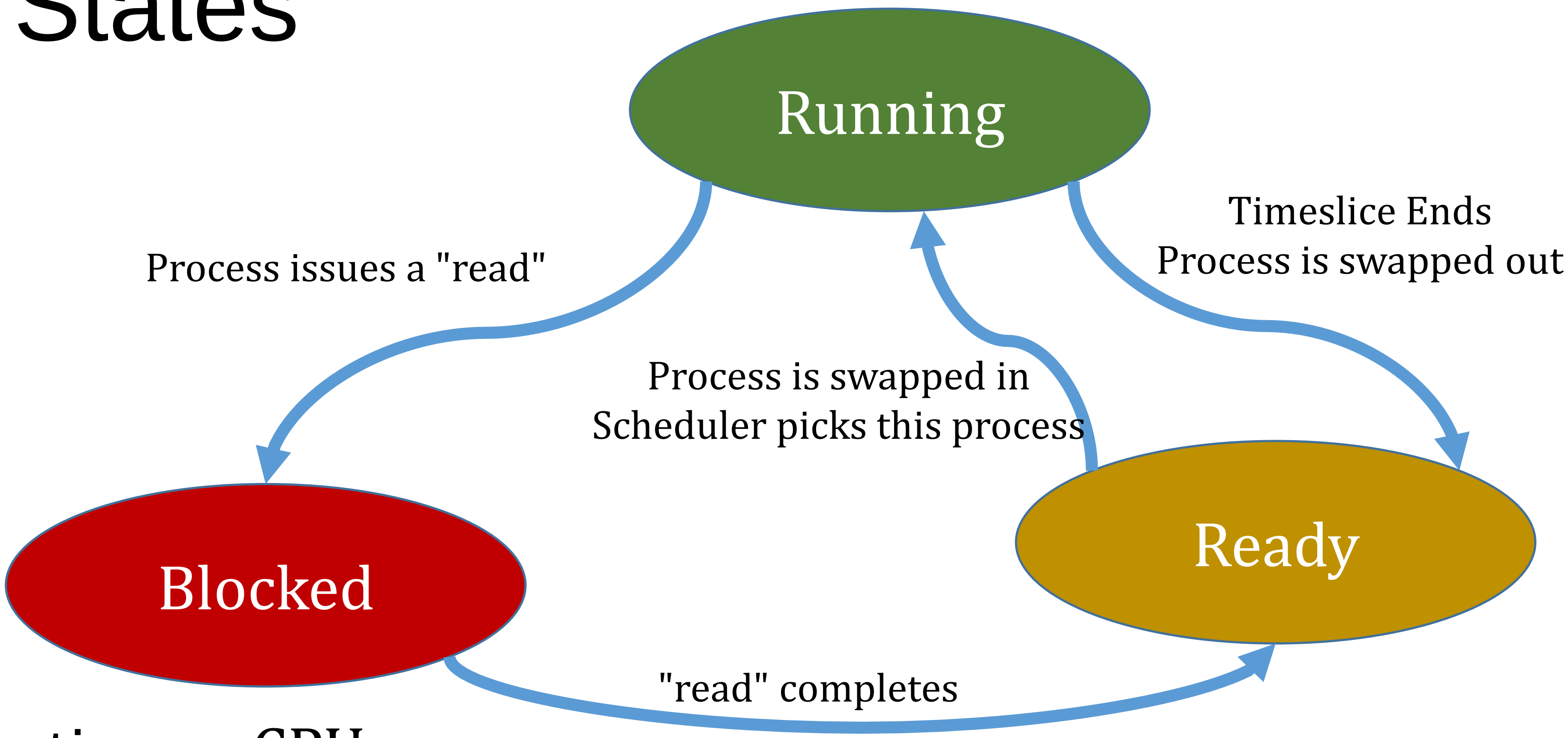
Side Topic: fork() and virtual memory

- When a fork occurs, active pages are tagged as belonging to BOTH parent and child processes
- If modification of that page is required, it must be duplicated and tagged as either a child or parent page
- Most of the time, exec will overwrite all pages, and they will revert to parent only pages

Time Sharing – Concurrence on CPU



Process States



- Running: executing on CPU
- Ready: Could run if CPU were available
- Blocked: Waiting for some event to occur

Tracing Process States (CPU intensive)

Time	PID 4879	PID 5321	Notes
1	Running	Ready	Scheduler chose 4879
2	Running	Ready	Time slice is 2 ticks, 4879 swapped out
3	Ready	Running	Scheduler chose 5321, 5321 swapped in
4	Ready	Running	for 2 ticks, then 5321 swapped out
5	Running	Ready	Process 4879 exits, swapped out
6	--	Running	Scheduler chose 5321
7	--	Running	Time slice is 2 ticks, Process 5321 exits
8	--	--	

Tracing Process States (I/O intensive)

Time	PID 4879	PID 5321	Notes
1	Running	Ready	Scheduler chose 4879, 4879 initiates I/O
2	Blocked	Running	4879 swapped out, 5321 swapped in
3	Blocked	Running	Time slice is 2 ticks, 5321 still running
4	Blocked	Running	No one else is waiting, 5321 can still run
5	Ready	Running	IO for 4879 completes, 5321 exits
6	Running	--	Scheduler chose 4879
7	Running	--	4879 initiates another I/O, and is swapped out
8	Blocked	--	Nothing running… CPU idle

Data kept by OS (Kernel) for a Process

Process Management	Memory Management	File Management
Process ID	Pointer to Text Segment	Root directory
Parent Process	Pointer to Data Segment	Working directory
Process Group	Pointer to Stack Segment	File Descriptors
Process State		User ID
Priority		Group ID
Scheduling Parameters		
Registers		
Instruction Pointer		
Stack Pointer	See task_struct in Linux Source Code	
Signals		
Time started		
…		

UNIX Process Info

- "ps" command
  - Standard process attributes
- /proc directory
  - If you have root privilege, more interesting information
- "top" command
  - CPU/Memory usage statistics with "top" processes identified

Orphan process

- Parent process are responsible for "reaping" the status of a child process
  - OS keeps child process alive, even after exit, to maintain exit status
- If a parent dies before waitpid completes for a child, the child becomes an *orphan* process
  - The "init" process (pid=1) becomes the parent of the orphan process
- For an example, run examples/process/orphan
  - Do a "ps –l" to see the result… look at the parent pid
  - Remember to kill the orphan: "kill -9 <pid>"

Zombie Processes

- When a child dies, a SIGCHILD signal is sent to the parent
- If the parent doesn't wait and the child exits, it becomes a "Zombie" (status of "Z" in ps)
  - The child is dead, but OS keeps it's status alive
- Zombie processes are around until either the parent calls wait() or waitpid(), Or the parent exits (when the Zombie becomes an orphan)

- OS Keeps integer status, but frees up all other process resources

Using "man" pages

- The "man" command accesses the on-line UNIX documentation
- Optionally, specify which "page" to look at by specifying a numeric argument to the "man" command
  - Page 2: system calls
  - Page 3: Library functions
- E.g. "man 2 exec" or "man 3 execlp"
- Or go on-line to: http://man7.org/linux/man-pages/index.html