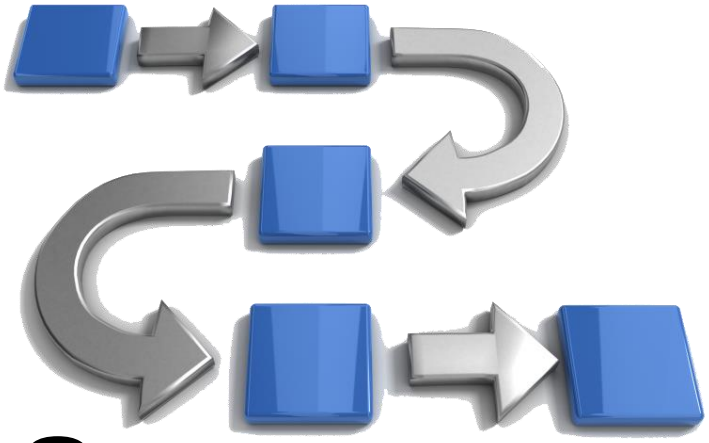


Using Processes

Computer Systems Chapter 8.2, 8.4

Abstract View

When I run my program, it has access to the entire computer, including the processor, memory, keyboard, display, disk drives, network connections, etc. etc. etc.

Leaky Abstraction

- In fact, most hardware supports multiple concurrent users
- Each user is often running multiple programs concurrently
- System services (called “deamons”) are often running to provide real-time capabilities
- Even running on a multi-core machine, the number of concurrently running programs almost always exceeds the number of processors.

What is a Process?

An invocation of a program

- Created by C library call “fork”
- Process ID: a numeric identifier associated with a process (PID)
  - Four digits long… 0001 to 9999
- Ended by “exit” library call (in stdlib.h) or “kill” interrupt
  - Return from main returns to a function which issues an exit
- Process issues a return code
- Process is removed from the system when return code is read

Process Hierarchy

- New processes can only be created by existing processes
  - The creator is called the ***parent process*** or "ppid"
  - The spawned process is called a ***child process***
- Parent processes are responsible for their children
- In UNIX, when you boot the machine, a root process is created
  - Root process spawns a login monitor process
  - When you log in, login monitor creates a terminal (shell) process
  - When you type a command in a shell process, it creates a command process.

Listing Processes

- In UNIX, the "ps" command lists processes
- By default, "ps" lists your process and all of it's children
- To list all processes owned by you, "ps –u *userid*"
- To list all processes by all owners on this machine, "ps –e"

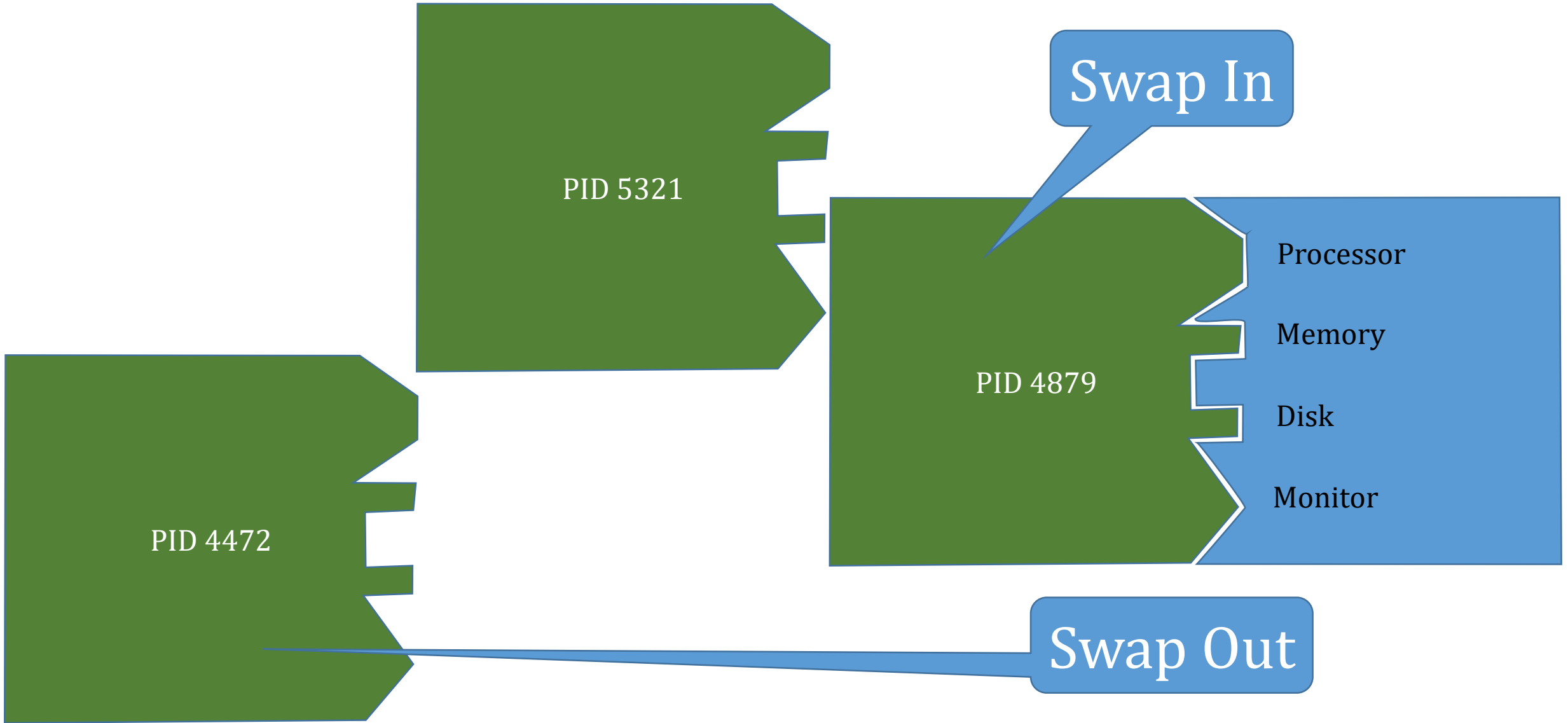
```
alpha:~/CS220> ps
 PID      TTY       TIME      CMD
 2933     pts/3     00:00:00 tcsh
 3057     pts/3     00:00:00 ps
```

```
alpha:~/CS220> ps -utbartens
 PID      TTY       TIME      CMD
 2836     ?         00:00:00 sshd
 2837     ?         00:00:00 tcsh
 2839     ?         00:00:00 sftp-server
 2913     ?         00:00:00 sshd
 2914     ?         00:00:00 tcsh
 2923     ?         00:00:00 sftp-server
 2932     ?         00:00:00 sshd
 2933     pts/3     00:00:00 tcsh
 3058     pts/3     00:00:00 ps
```

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 Context

- There is information/data associated with each process
  - Register values
  - Values in memory
  - How much data has been read from a file
  - etc.
- The sum of all state for the entire process is called the process *context*
- When a process is active, it has access to its entire context

Time Slicing Issue – Context Swap

- When a process is swapped out, we must save it's context
- When a process is swapped in, we must load it's context
- The process of saving the outgoing context, and loading the incoming context is called a "context swap"
- Context swapping is "overhead" – extra resource needed that does not do the processes work
  - No context swapping required for batch jobs

Process Swapping / Context Switch

- Wait for Instruction to End
- Save context of swap out process
  - Registers (especially %rip) & flags
  - Main Memory (stack and heap)
  - I/O status
- Restore swapped in context
  - Registers and Memory and I/O status
- Restart instruction processing cycle

Swapping Memory

Bad Idea:

Write Swap Out address space from memory to disk

Read Swap In address space from disk to memory

- A 32 bit address space is 4G, 64 bit address space is huge
- Writing 4G to disk takes ~1G/sec or 4 seconds
- Times slices are MUCH smaller than 1 second
- You would spend 99.9999% of the time reading/writing memory!

Solution: Paged Memory

Operating System Process Status Table

- Keeps track of every process
- Process added to OS process status table when a parent spawns a child process
- The child process is alive (running) as long as it continues to execute instructions
- When a child exits (or is killed), it becomes "dead", **but it is still in the process table! It is now a "zombie"**
  - Process table holds the return code from the process
- Process is removed from the OS process table when the parent "reaps" the process (reads the child's return code)

Forks

When you "fork" a single process…

- A new process is created… a ***child*** of the existing process
- The process doing the forking is the ***parent*** process

- At the point of the fork, the parent's context is cloned
  - The child gets a FULL COPY of the parent's address space
  - The child inherits a copy of the parent's IO resources
  - The parent's register values are cloned, including %rip!

What does “clone” mean?

- Start out identical…

- … but as time goes by, clones diverge…

After the Fork

- Two independent copies of memory that start out identical, but diverge as parent and child write different things in their memory
- Two independent copies of IO resources that start out pointing to single IO resources, but may diverge as parent and child manipulate these resources independently
- Two independent copies of Register values that start out identical, but diverge as parent and child write different values
- **No communication between parent and child through memory!**
- Parent is still responsible for child.

How can you tell child from parent?

- Memory is cloned… parent and child are the same
- Register values are cloned… parent and child are the same
  - Same %rip implies the same instruction(s) are executing
- The ONLY difference between parent and child is the return value from the “fork” function
  - %rax register is different!
  - For the parent, the “fork” function returns the PID of the child
  - For the child, the “fork” function returns zero (0)
    - Zero is not a valid PID

fork standard library call

```
#include <unistd.h>

pid_t pid;
…
pid = fork();
if (pid==0) { // This is the child
…
} else { // This is the parent… pid is the child pid
…
}
```

Only parent executes…
No child yet.

Both parent and child execute after fork

Only the child executes if pid==0

Only the parent executes if pid!=0

Cleaning Up After Your Kids

- When a child process exits, it posts its return code
  - BUT IT STAYS ACTIVE
  - Must stay active until it's parent process reaps the child's return code
- Parent must read the return code from its children
  - Reading the return code is called *reaping* the child process
  - When a child process has been reaped, it can be removed from OS tables
  - Reaping a child process can be done with either "wait" or "waitpid" C system library calls
    - "wait" – allows you to reap any child process
    - "waitpid" – allows you to reap a specific child process
  - Both "wait" and "waitpid" make parent go idle until child exits

"Automatic" clean-up

- C "exit" processing performs automated clean-up:
  - closes any files you have left open
  - free's any space you have malloc'ed
  - waits for any unreaped children

- Automatic clean-up is frowned on!
  - What happens if you never get there?
  - It might take days before the parent exits
  - It's messy – you know when you are done with a resource better than the OS

Loading and Running Programs

int execve(char * filename, char *argv[], char *envp[])

- Library function in unistd.h
- filename – Name of ELF executable file
- argv –> Null terminated array of arguments
- envp –> Null terminated array of environment variables

- Loads executable from filename
- Calls “main” function, but sets return value to OS
- Never returns to calling code! (unless error occurs loading)

Over-simplified "Shell"

```
char cbuf[256];
pid_t cpid; int cstat;
while(fgets(cbuf,sizeof(cbuf),stdin)) {
        cpid=fork();
        if (cpid==0) { execve(qfile(cbuf),qargs(cbuf),NULL); }
        waitpid(cpid,&cstat,NULL);
}
exit(0);
```

Read command from stdin

child pid loads and executes command
Note: never returns!

parent (shell) pid waits for command to finish