CS 201

Processes

Gerson Robboy
Portland State University

Review

Definition: A *process* is an instance of a running program.

- One of the most fundamental concepts in computer science.
- Not the same as "program" or "processor"

A *program* is a set of instructions and initialized data in a file, usually found on a disk.

A *process* is an instance of that program while it is running, along with the state of all the CPU registers and the values of data in memory.

A single program can correspond to many processes; for example, several users can be running a shell.

Processes

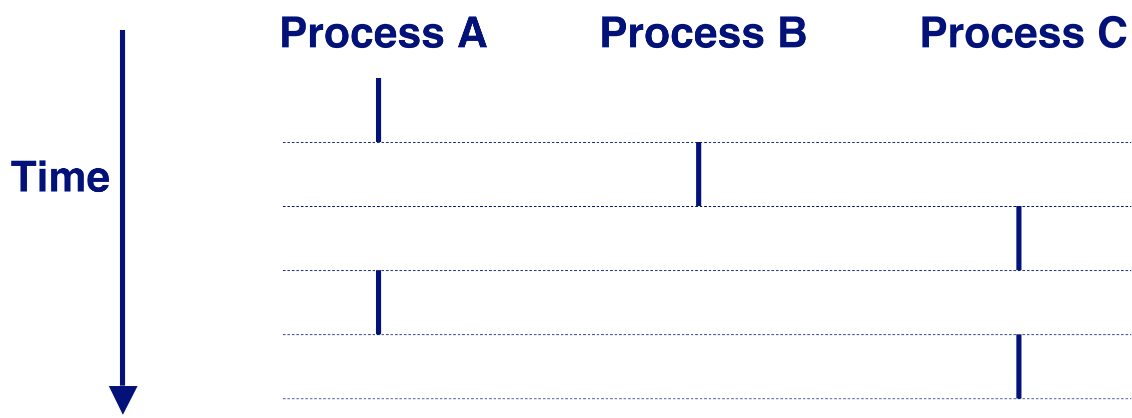
The operating system provides each process with a *virtual machine*

If a process were a thing that can have a point of view, it would see itself having exclusive use of the computer.

- Running continuously on the CPU
- In possession of the entire memory space, CPU registers, and I/O devices
- No other processes are visible
- If it checked the wall time often, the process might notice gaps in time it can not account for

Logical Control Flows

Each process has its own logical control flow

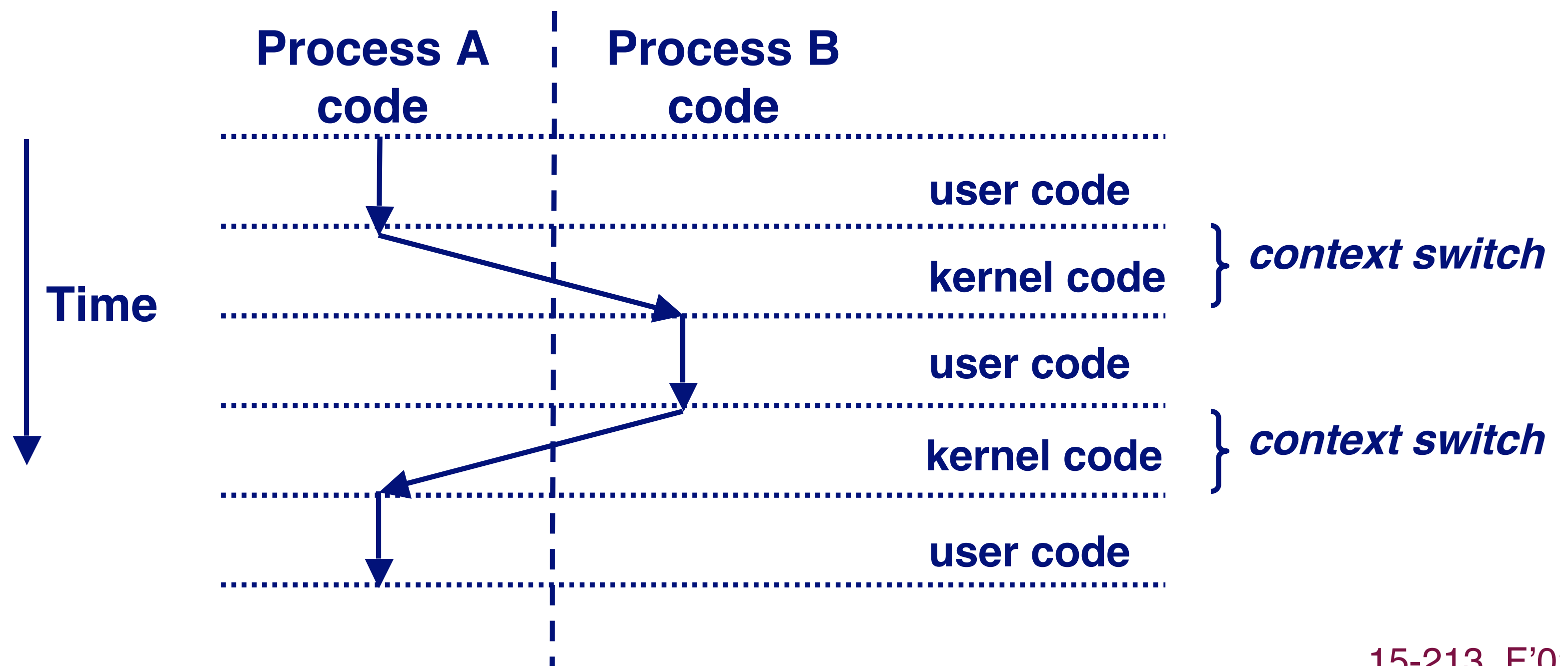


Context Switching

Processes are managed by a shared chunk of OS code called the *kernel*

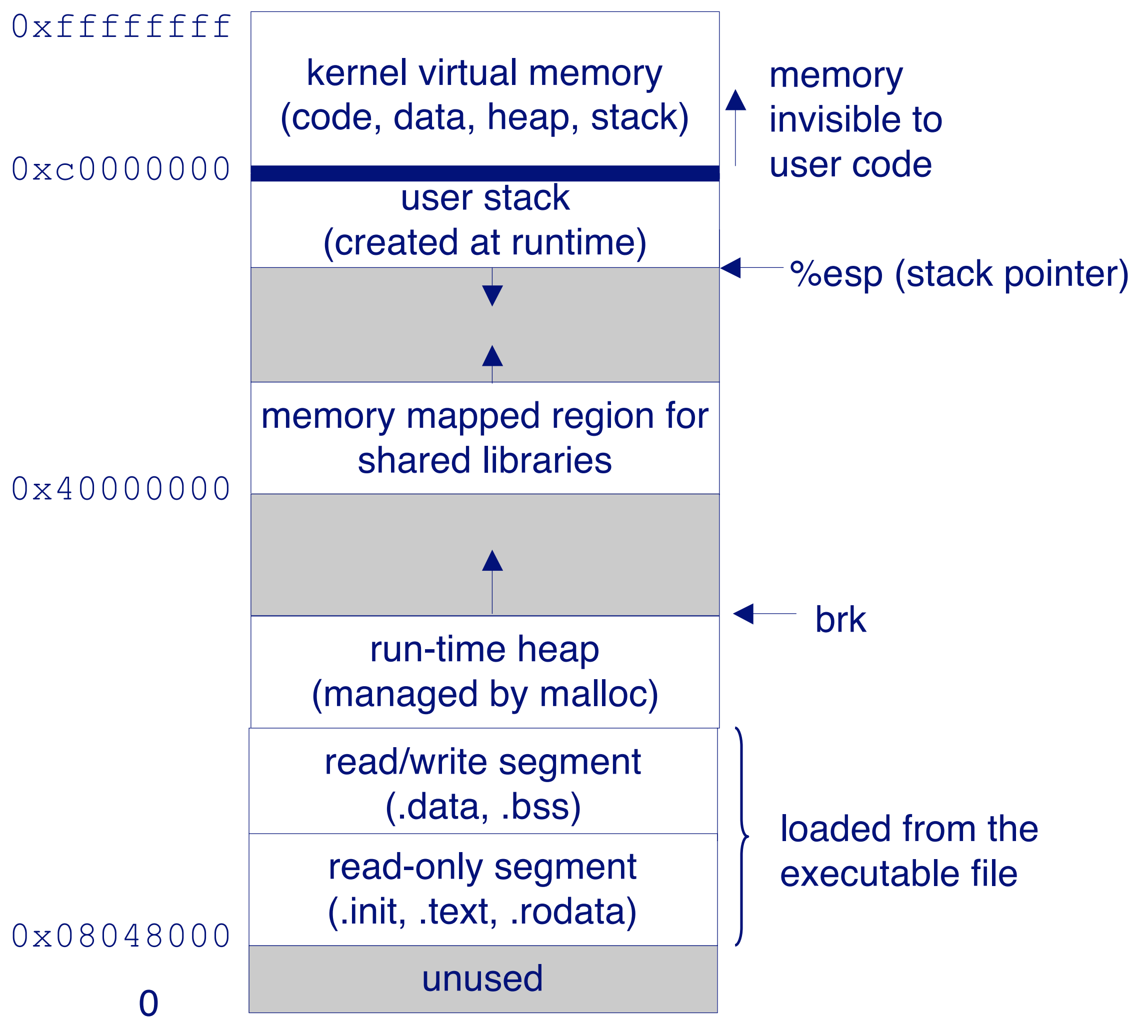
- The kernel is not a separate process, but rather runs as part of some user process

Control flow passes from one process to another via a *context switch*.



Private Address Spaces

Each process has its own private address space.



How do processes get created?

The *fork()* system call creates a new process.

Every process is created by another process.

- **With one exception, the very first process…**

fork() creates a duplicate of the process that called it.

`fork`: Creating new processes

`int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- Fork is called once but returns in two separate processes.
- The processes are identical except for one detail:
  - fork returns 0 to the child process
  - fork returns the child's `pid` to the parent process

int fork(void)

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

In this code example, what will you see on the screen?

Fork

Key Points

- Parent and child both run the same code
  - Distinguish parent from child by return value from `fork`
- Both processes, after fork, have identical state
  - Including shared open file descriptors
  - Relative ordering of their print statements undefined
  - The two processes will go their separate ways without synchronizing
- This is important: Separate memory spaces.

Fork Example #1

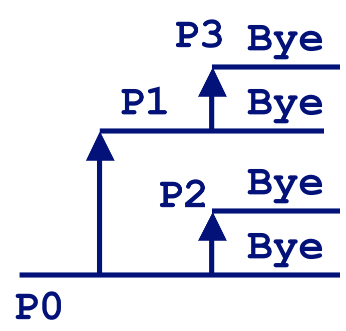
What does this program do?

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Exercise

What does this program print?

```
void doit()
{
    fork();
    fork();
    printf("hello\n");
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

Exercise: What does this program print?

```
static int j = 0;

do_child(int i)
{
  int pid;
  if (i < 2){
        pid = fork();
        if(pid == -1)
            exit(0);
        else if(pid == 0){
            do_child(i+1);
        } else {
            j++;
            printf("This is process %d, j=%d\n", i, j);
        }
  } else {
        j++;
        printf("This is process %d, j=%d\n", i, j);
  }
}

main()
{
  do_child(0);
}
```

OK, so now we know how to create processes.

Doesn't a computer do something besides run duplicate copies of what's already running?

How?

`exec`: Running new programs

A family of related functions: execv, execp, execl

`int execl(char *path, char *arg0, char *arg1, …, 0)`

- **loads and runs executable at `path` with args `arg0`, `arg1`, …**
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
  - "real" arguments to the executable start with `arg1`, etc.
  - list of args is terminated by a `(char *)0` argument

Here's what they all do:

- **Overwrite the calling *process* with a new *program***
  - Does not create a new process
  - Runs a new program
- **returns `-1` if error, otherwise doesn't return**
  - Why doesn't it return?

Example

A program that creates a child process, the child executes */usr/bin/ls*, and then the parent prints "done."

```
main() {
   if (fork() == 0) {
      execl("/usr/bin/ls", "ls", 0);
   }
   wait(0);   // This is the parent
   printf("done\n");
   exit(0);
}
```

`exit`: Destroying Process

`void exit(int status)`

- exits a process
  - Normally return with status 0
- `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

`wait`: Synchronizing with children

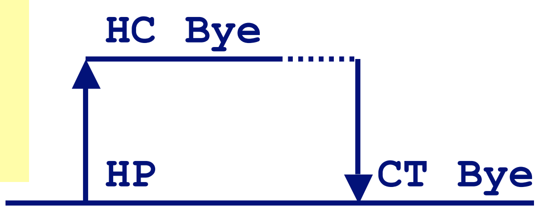
```
int wait(int *child_status)
```

- suspends current process until one of its children terminates
- return value is the `pid` of the child process that terminated
- If the child has already terminated, then wait returns its pid immediately
- If `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

`wait`: Synchronizing with children

```
void fork9() {
   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();
}
```



`wait`: reaping children

```
int wait(int *child_status)
```

- If the child has already terminated, then wait returns its pid immediately
- What if many children have terminated?
  - Will wait reliably return pids of all terminated children?
  - Is it possible to lose some?
- Terminated child processes turn into zombies
  - Wait *reaps* the zombies

Still more on wait

What ‘wait’ is really waiting for is a SIGCHLD signal.

Other signals can also cause wait to return.

When wait returns, check to see if it really returns the pid of a child process

- Otherwise, it could have been some other signal

Multitasking

The System Runs Many Processes Concurrently

State consists of memory image + register values

- general registers
- system registers include program counter, pointer to page tables, …

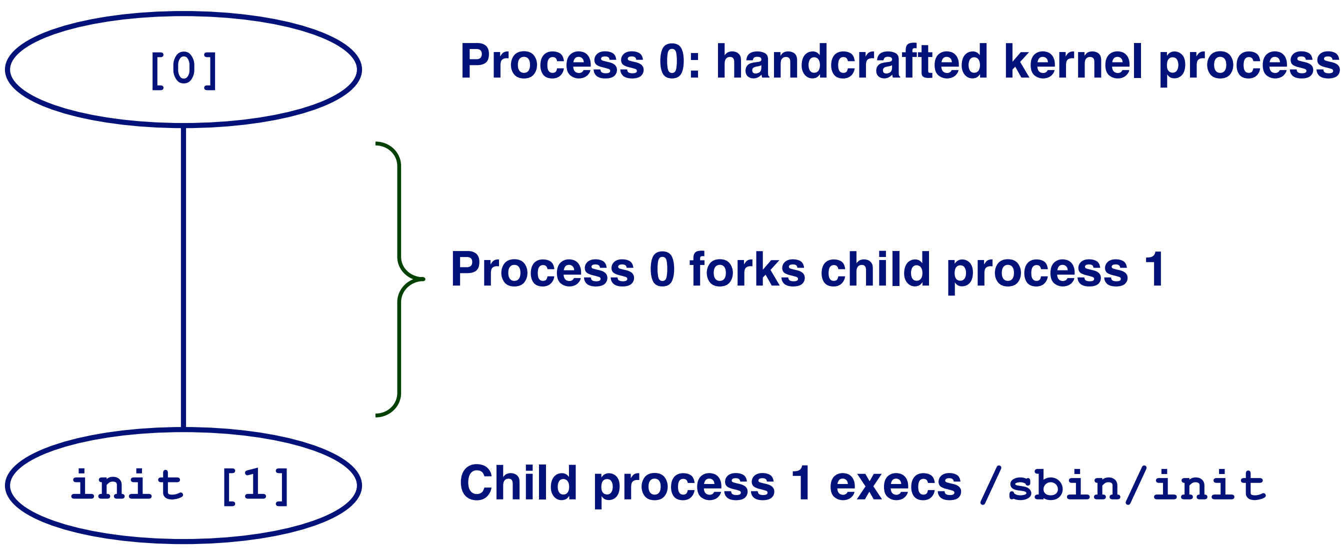
The kernel continually switches from one process to another

- Sometimes a process blocks waiting for I/O
- Sometimes the timer pre-empts a process

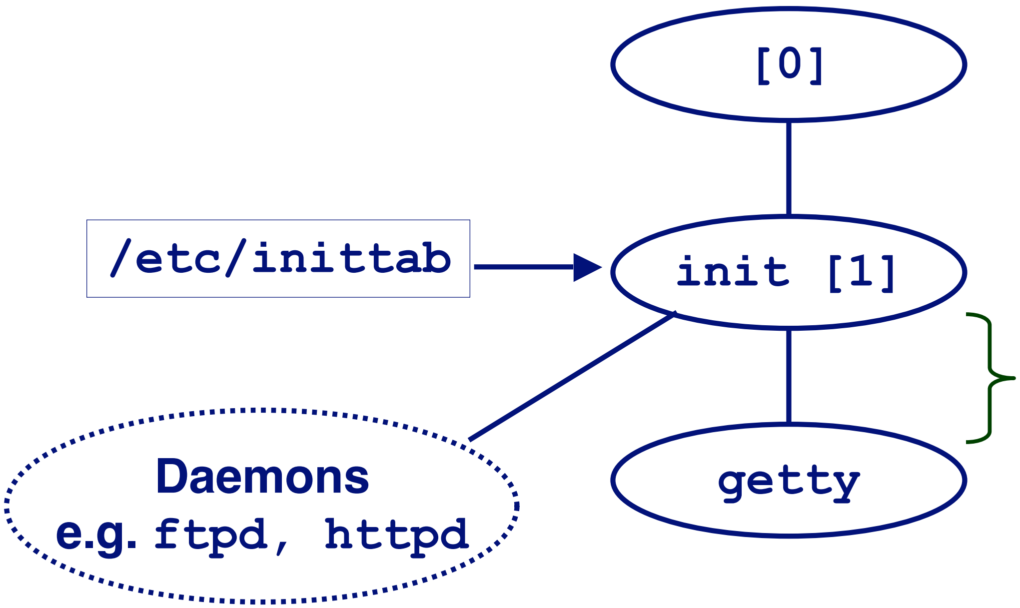
To us, observing from outside the computer, it appears that all processes are running concurrently.

Unix Startup: Step 1

1. Pushing reset button loads the `PC` with the address of a small bootstrap program.
2. Bootstrap program loads the operating system kernel from the file system, or maybe a secondary bootstrap program
3. Bootstrap program passes control to kernel.
5. Kernel handcrafts "process 0."

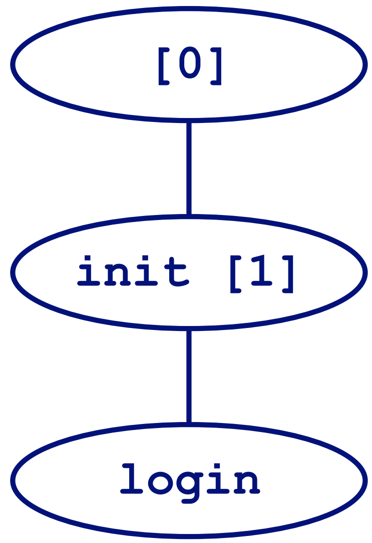


Unix Startup: Step 2



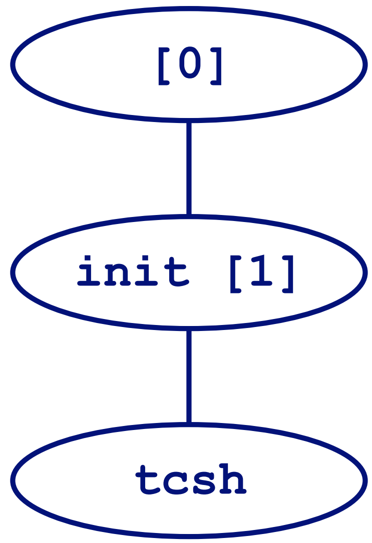
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

Unix Startup: Step 3



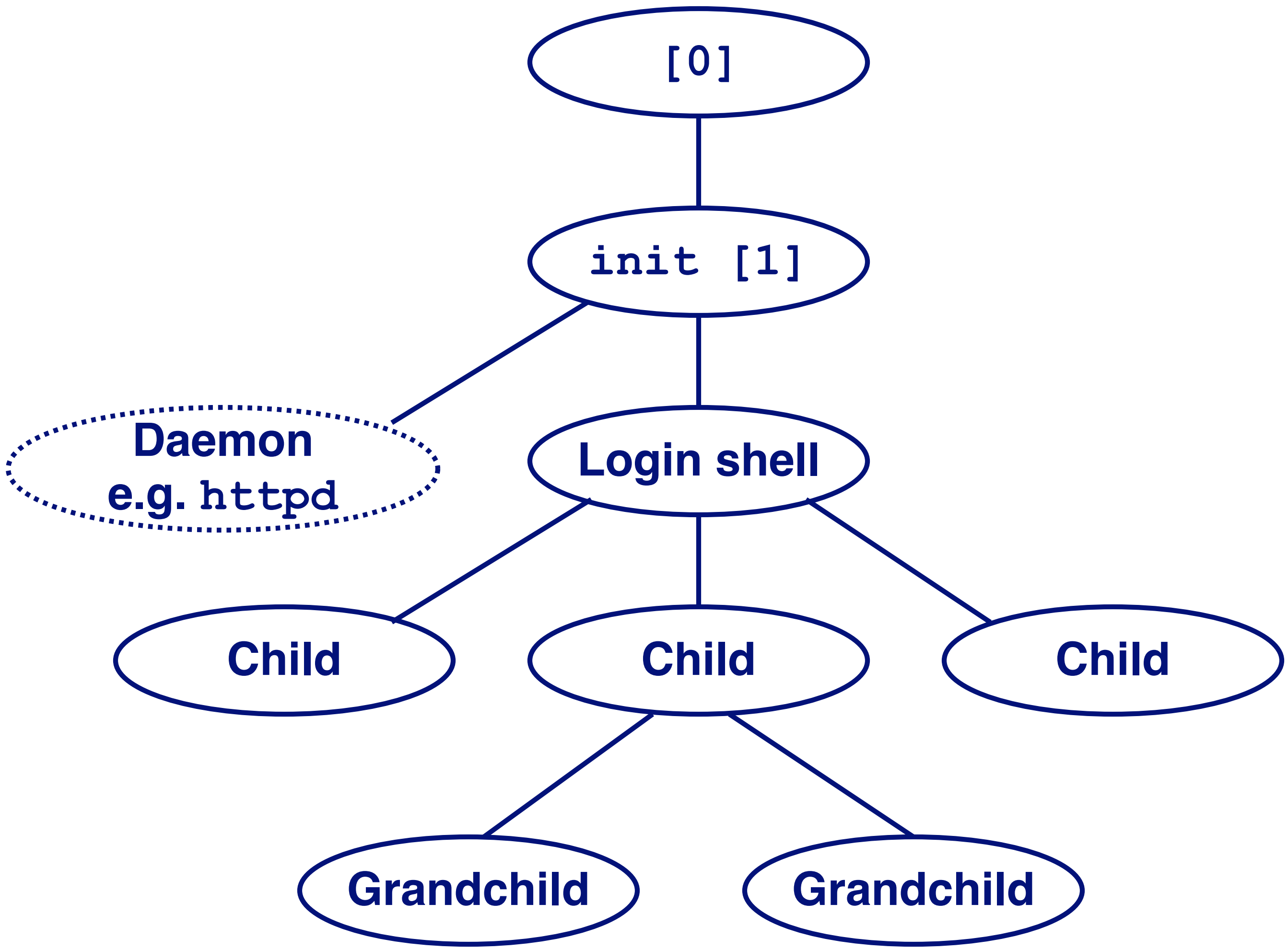
The `getty` process execs a `login` program

Unix Startup: Step 4



`login` **reads login and passwd.**
if OK, it execs a *shell.*
if not OK, it execs another `getty`

Unix Process Hierarchy



So the kernel never spontaneously creates a process, except for process zero.

The kernel creates a process when some existing process calls fork().

Programmer's Model of Multitasking

Basic Functions

- `fork()` spawns new process
  - Called once, returns twice
  - Parent and child process both resume running where fork() returns.
- `exit()` terminates the process that calls it
  - Called once, never returns
- `wait()` and `waitpid()` wait for and reap terminated children
- `execl(), execv(), and friends`
  - run a new program in an existing process
  - Called once, normally never returns

Example: Shell Programs

A *shell* is an application program that runs programs on behalf of the user.

- `sh` – Original Unix Bourne Shell
- `csh` – BSD Unix C Shell, `tcsh` – Enhanced C Shell
- `bash` –Bourne-Again Shell

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
       /* read */
       printf("> “);
       fgets(cmdline, MAXLINE,
stdin);
       if (feof(stdin))
           exit(0);
       eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

Summarizing

Processes

- At any given time, a system can have multiple active processes
- Only one can execute at a time, though
  - Per CPU, that is
- Each process, from its own point of view, appears to have total control of a virtual computer
  - A CPU, including its registers
  - A virtual memory space

Summarizing (cont.)

Spawning Processes

- Call to `fork`
  - One call, two returns

Terminating Processes

- Call `exit`
  - One call, no return

Reaping Processes

- Call `wait` or `waitpid`

Replacing Program Executed by Process

- Call `execl` (or variant)
  - One call, normally no return