CS 105
"Tour of the Black Holes of Computing!"

Processes

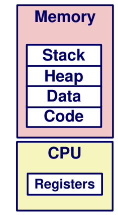
Topics
- Process context switches
- Creating and destroying processes

Processes

Def: A *process* is an instance of a running program
- One of the most profound ideas in computer science
- Not the same as "program" or "processor"

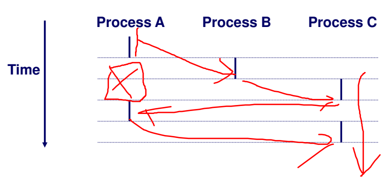
Process provides each program with two key abstractions:
- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private address space
  - Each program seems to have exclusive use of main memory

How are these illusions maintained?
- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system

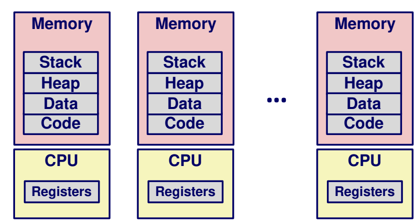


Logical Control Flows

Each process has its own logical control flow

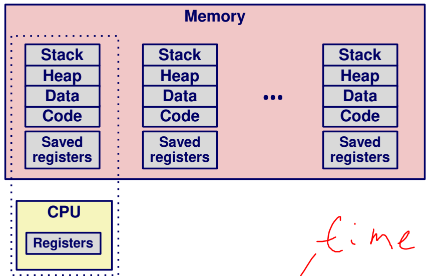


Multiprocessing: The Illusion



Computer runs many processes simultaneously
- Applications for one or more users
  - Web browsers, email clients, editors, …
- Background tasks
  - Monitoring network and I/O devices
  - Web and mail servers, VPN management, auto-backups, Skype, …

Multiprocessing: The (Traditional) Reality

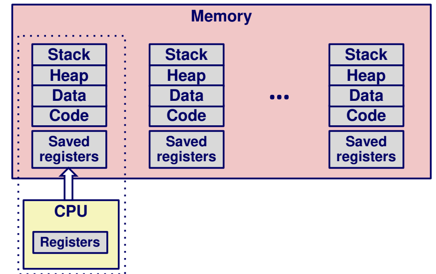


time slicing

Single processor executes multiple processes concurrently

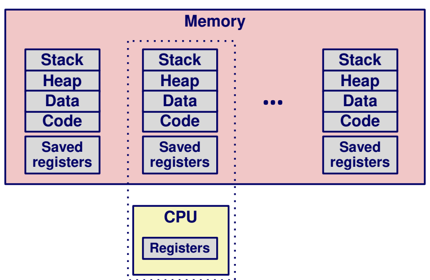
- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system (later in course)
- Nonexecuting processes' register values saved in memory

Multiprocessing: The (Traditional) Reality



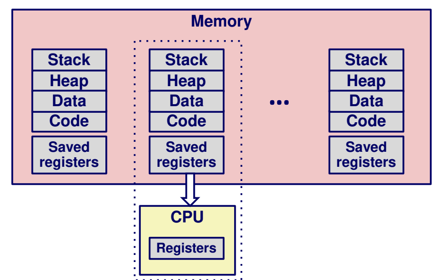
Save current registers in memory

Multiprocessing: The (Traditional) Reality



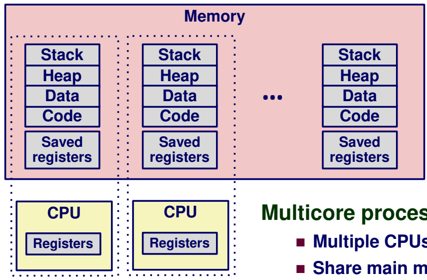
Schedule next process for execution

Multiprocessing: The (Traditional) Reality



Load saved registers and switch address space (context switch)

Multiprocessing: The (Modern) Reality



Multicore processors

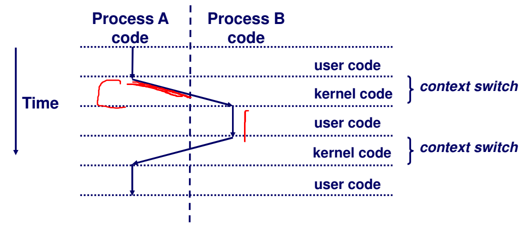
- Multiple CPUs on single chip
- Share main memory (and some of the caches)
- Each can execute a separate process
  - Scheduling of processors onto cores done by kernel

Context Switching

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

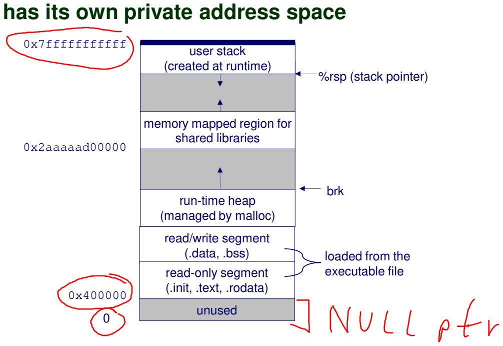
- Important: the kernel is not a separate process, but rather runs as part of (or on behalf of) some user process

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



Private Address Spaces

Each process has its own private address space



System-Call Error Handling

On error, Unix system-level functions typically return -1 and set global variable `errno` to indicate cause.

Hard and fast rule:

- You **MUST** check the return status of *every* system-level function!!!
- Only exception is the handful of functions that return `void`

Example:

```
pid = fork();
if (pid == -1) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(1);
}
```

Error-Reporting Functions

Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(1);
}
```

(Aborting on error is generally bad idea but handy for demo programs)

```
if ((pid = fork()) == -1)
    unix_error("fork error");
```

Note: assignment inside conditional is bad style but common idiom

Error-Handling Wrappers

We simplify the code we present to you even further by using Stevens-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork())  == -1)
        unix_error("Fork error");
    return pid;
}
```

```
pid = Fork();
```

Lousy approach in real life but useful for simplifying examples

Obtaining Process IDs

Every process has a numeric *process ID* (PID)

Every process has a parent

`pid_t getpid(void)`

- Returns PID of current process (self)

`pid_t getppid(void)`

- Returns PID of parent process

Process States

From a programmer's perspective, we can think of a process as being in one of three states:

Running

- Process is either executing or waiting to be executed, and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

Stopped

- Process execution is *suspended* and will not be scheduled until further notice (future lecture when we study signals)

Terminated

- Process is stopped permanently (due to finishing or serious error)

Terminating Processes

Process becomes terminated for one of three reasons:

- Receiving a signal whose default action is to terminate (future lecture)
- Calling the `exit` function
- Returning from the `main` routine (which actually calls `exit` internally)

`void exit(int status)`

- Terminates with an *exit status* of `status`
- Convention: normal return status is 0, nonzero on error (Anna Karenina)
- Another way to explicitly set the exit status is to return an integer value from the main routine

1-127
1="normal"
2="usage"

`exit` is called once but never returns.

Creating Processes: `fork()`

Parent process creates a new running *child process* by calling `fork`

`int fork(void)`

- Returns 0 to the child process, child's PID to parent process
- Child is *almost* identical to parent:
  - Child get an identical (but separate) copy of the parent's virtual address space.
  - Child gets identical copies of the parent's open file descriptors, signals, and other system information
  - Child has a different PID than the parent

`fork` is interesting (and often confusing) because it is called *once* but returns *twice*

Huh? Run that by me again!

`fork` Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
       /* Child */
       printf("child : x=%d\n", ++x);
       exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
                                    fork.c
```

x = 1

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdin, stdout, stderr` are *the same* in both parent and child

Important!!!

Modeling `fork` with Process Graphs

A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:

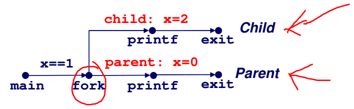
- Each vertex is the execution of a statement
- a → b means a happens before b
- Edges can be labeled with current value of variables
- `printf` vertices can be labeled with output
- Each graph begins with a vertex with no incoming edges

Any *topological sort* of the graph corresponds to a feasible total ordering.

- Total ordering of vertices where all edges point from left to right

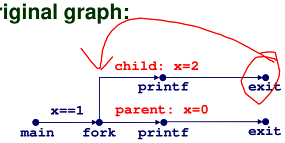
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

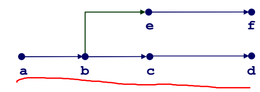


Interpreting Process Graphs

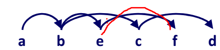
Original graph:



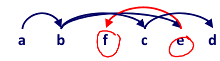
Relabeled graph:



Feasible total ordering:

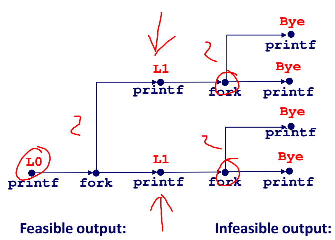


Infeasible total ordering:



`fork` Example: Two consecutive `fork`s

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

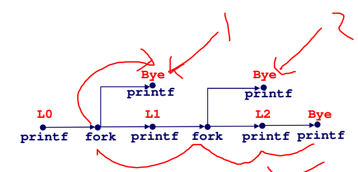


Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

`fork` Example: Nested `fork`s in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

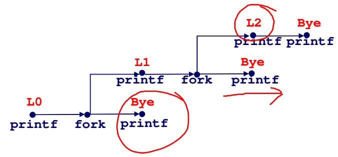


Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

fork Example: Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

Reaping Child Processes

Idea

- When process terminates, it still consumes resources
  - Examples: exit status, various OS tables
- Called a "zombie"
  - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child (using `wait` or `waitpid`)
- Parent is given exit status information
- Kernel then deletes zombie child process

What if parent doesn't reap?

- If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
- So, only need explicit reaping in long-running processes
  - e.g., shells and servers

Zombie Example

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 bash
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 bash
 6642 ttyp9    00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped

Nonterminating Child Example

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

wait: Synchronizing with Children

Parent reaps a child by calling the `wait` function

int* stat;
wait(stat)

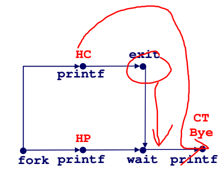
`int wait(int *child_status)`

- 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 value that tells why child terminated and gives its exit status:
  - Checked using macros defined in `wait.h`
    - » `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
    - » See textbook for details

int stat;
wait(&stat)

wait: Synchronizing with Children

```
void fork9() {
    int child_status;

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



Feasible output:	Infeasible output:
HC	HP
HP	CT
CT	Bye
Bye	HC

Another Wait Example

- If multiple children completed, will take in arbitrary order
- Can use WIFEXITED and WEXITSTATUS to probe status

```
void fork10()
{
    pid_t pid[N];
    int I, child_status;
    for (i = 0; i < N; i++) {
       pid[i] = fork();
       if (pid[i] == 0)
           exit(100 + i); /* Child */
    }
    for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                  wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
    }
}
```


Waitpid

- `waitpid(pid, &status, options)` WNOHANG
  - Can wait for specific process
  - Various options available (see man page)

```
void fork11()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++) {
       pid[i] = fork();
       if (pid[i] == 0)
           exit(100 + i); /* Child */
    }
    for (i = 0; i < N; i++) {
       pid_t wpid = waitpid(pid[i], &child_status, 0);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                  wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
    }
}
```

exec: Running New Programs

```
int execlp(char *what, char *arg0, char *arg1, …, 0)
```

- Loads and runs executable at `what` with args `arg0`, `arg1`, …
  - `what` is name or complete path of an executable
  - `arg0` becomes name of process
    - » Typically `arg0` is either identical to `what`, or else contains only the executable filename from `what`
  - "Real" arguments to the executable start with `arg1`, etc.
  - List of args is terminated by a `(char *)0` argument
- Replaces code, data, and stack
  - Retains PID, open files, other system context like signal handlers
- Called once and never returns (except if there is an error)
  - Differs from `exit` because process keeps running, but program executed is brand-new

execlp Example

- Runs "`ls -lt /etc`" in child process
- Output is to `stdout` (why?)

```
main() {
   pid_t pid;
   int status;
   pid = fork();
   if (fork() == 0) {
      status = execlp("ls",
       "ls", "-lt", "/etc", NULL);
      if (status == -1) {
         fprintf(stderr, "ls: command not found\n");
         exit(1);
      }
   }
   wait(NULL);
   exit(0);
}
```


Summarizing

Processes

- At any given time, system has multiple active processes
- But only one (per CPU core) can execute at a time
- Each process appears to have total control of processor + private 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