Lecture 23

Processes

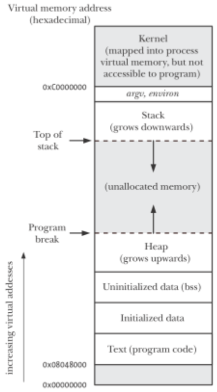
CPSC 275
Introduction to Computer Systems

Processes

- Definition: 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 virtual address space
    - Each program seems to have exclusive use of main memory
- How are these Illusions maintained?
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system

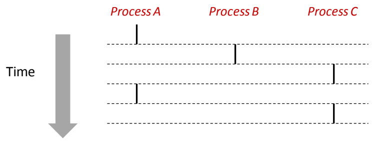
Memory Layout of a Process

- *Text segment*
  - machine-language instructions
  - Read-only
- *Initialized data segment*
  - global and static that are explicitly initialized
- *Uninitialized data segment*
  - global and static that are not explicitly initialized
  - Initialized to 0 when starting the program
- *Stack*:
  - Stores stack frames, one for each function called
- *Heap*
  - Dynamically allocated memory



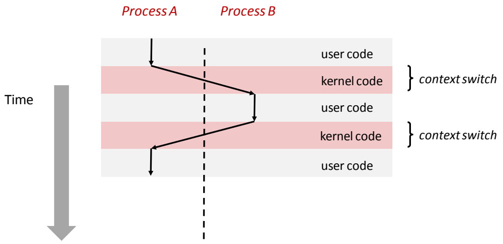
Concurrent Processes

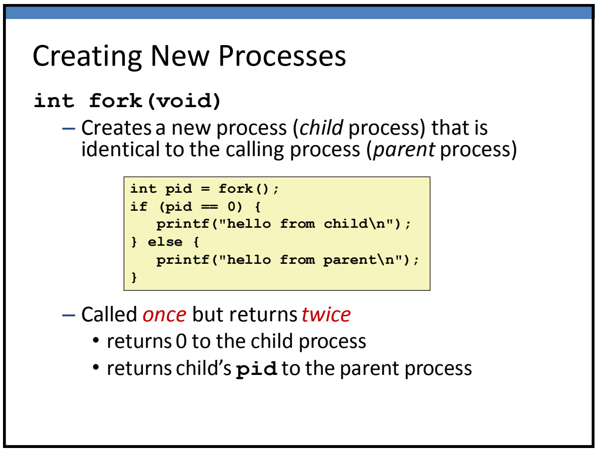
- Two processes *run concurrently* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



Context Switching

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





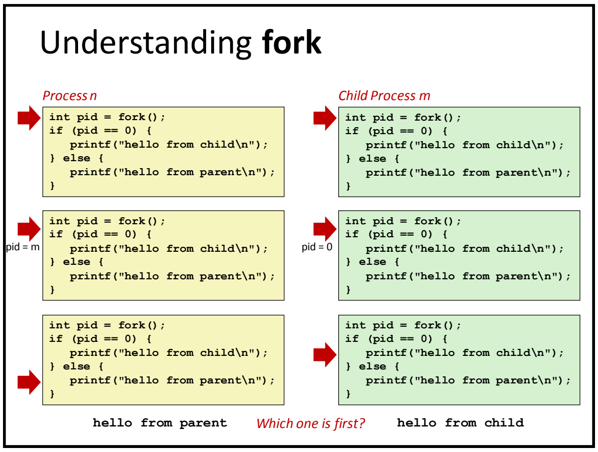
Creating New Processes

`int fork(void)`

- Creates a new process (*child* process) that is identical to the calling process (*parent* process)

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

- Called *once* but returns *twice*
  - returns 0 to the child process
  - returns child's **`pid`** to the parent process



Understanding **fork**

Process n

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

pid = m

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

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

Child Process m

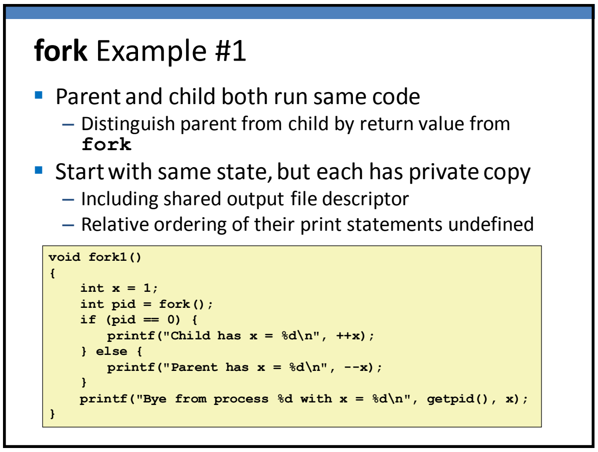
```
int pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

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

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

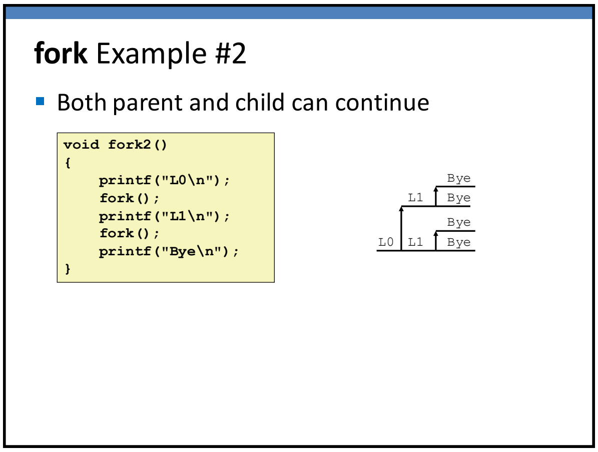
`hello from parent` *Which one is first?* `hello from child`



fork Example #1

- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

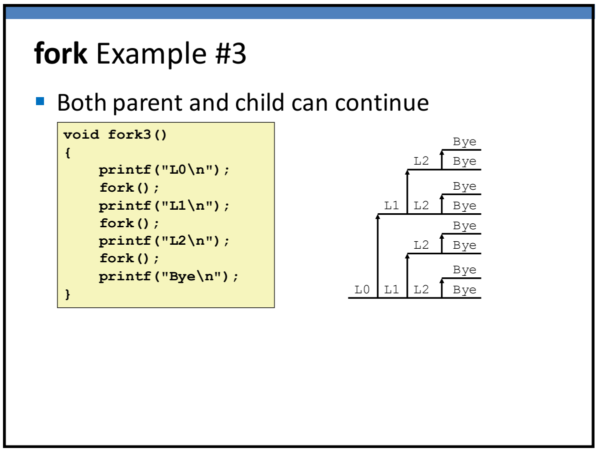
```
void fork1()
{
    int x = 1;
    int pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



fork Example #2

- Both parent and child can continue

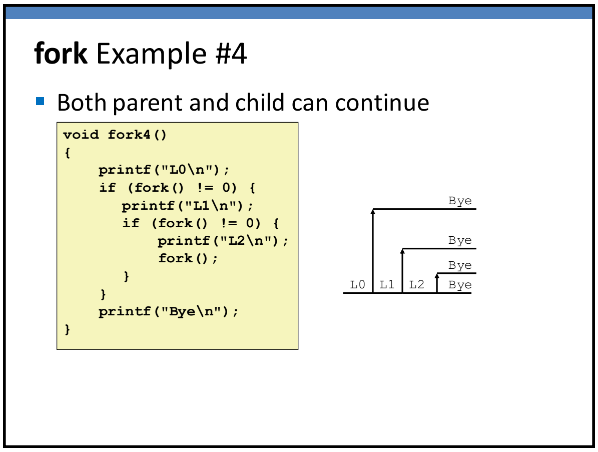
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



fork Example #3

- Both parent and child can continue

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



fork Example #4

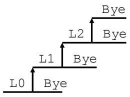
- Both parent and child can continue

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

fork Example #5

- Both parent and child can continue

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



Ending a 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);
}
```

Zombies

- Idea
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
- *Reaping*
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by `init` process

Zombie Example

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

Synchronizing with Children

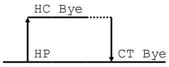
`int wait(int *child_status)`

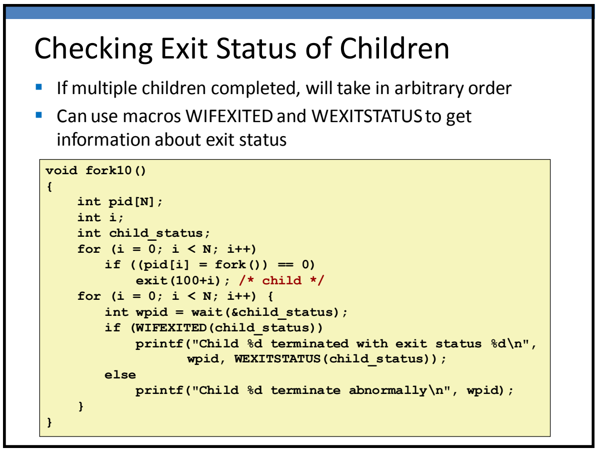
- suspends current process until one of its children terminates
- return value is the `pid` of the child process that terminated
- if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

Example: 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();
}
```

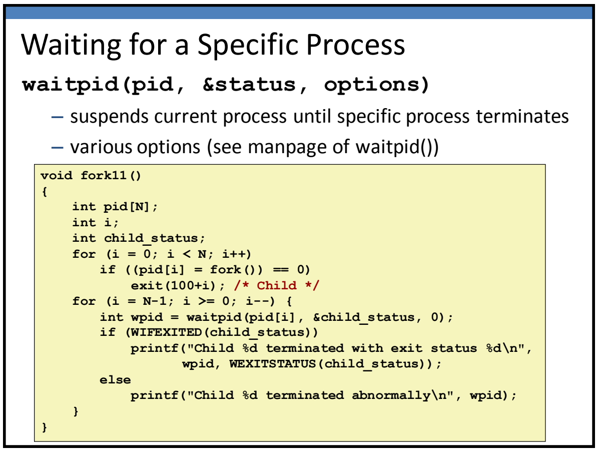




Checking Exit Status of Children

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    int pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* child */
    for (i = 0; i < N; i++) {
        int 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 terminate abnormally\n", wpid);
    }
}
```

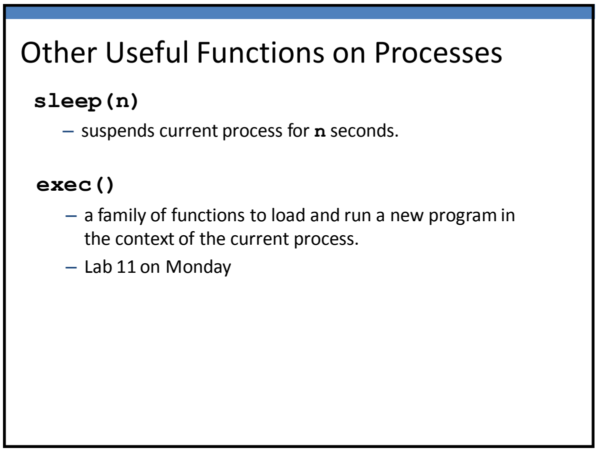


Waiting for a Specific Process

`waitpid(pid, &status, options)`

- suspends current process until specific process terminates
- various options (see manpage of waitpid())

```
void fork11()
{
    int pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        int 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);
    }
}
```



Other Useful Functions on Processes

`sleep(n)`

- suspends current process for **`n`** seconds.

`exec()`

- a family of functions to load and run a new program in the context of the current process.
- Lab 11 on Monday