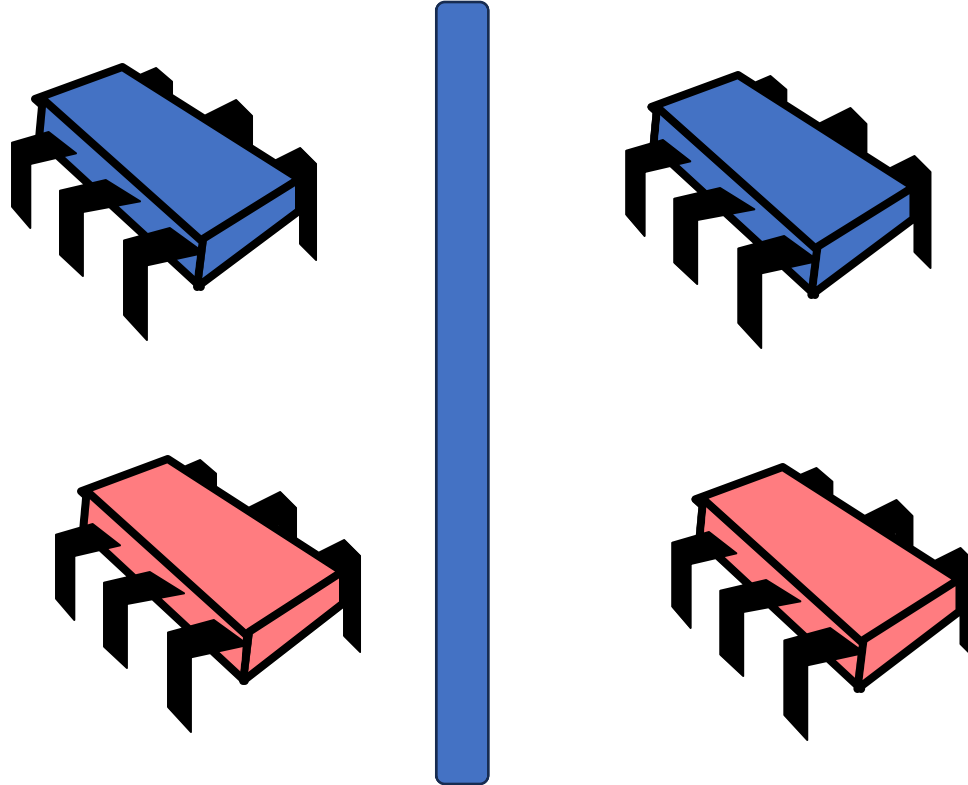
CSE113: Parallel Programming

- **Topics**:
  - Example Questions
  - Processes



Announcements

- HW 4 grades will be released this week.
- HW 5 is due today.
- SETs are out, please do them! It helps us out a lot.

Quiz

How many API calls do Barrier objects have?

- ◯ 0
- ◯ 1
- ◯ 2
- ◯ 3

Quiz

How many API calls do Barrier objects have?

- ◯ 0
- ◉ 1
- ◯ 2
- ◯ 3

Quiz

A barrier call emits which of the following events? Check all that apply

◯ barrier_lock
◯ barrier_arrive
◯ barrier_enqueue
◯ barrier_leave

Quiz

A barrier call emits which of the following events? Check all that apply

- ◯ barrier_lock
- ◉ barrier_arrive
- ◯ barrier_enqueue
- ◉ barrier_leave

Quiz

If a program uses both barriers and mutexes, the outcome is deterministic (i.e. the same every time) if there are no data conflicts.

◯ True
◯ False

Quiz

If a program uses both barriers and mutexes, the outcome is deterministic (i.e. the same every time) if there are no data conflicts.

◯ True
◉ False

If the mutex is protecting concurrent writes, there is non-determinism.

Quiz

Write a few sentences about what you think the best interface for parallel programming is, that is, do you think it is Atomics? Mutexes? Concurrent Data Structures? Barriers? Or even maybe the compiler should simply do it all automatically? Or is it some combination of the above? What are the trade-offs involved?

Sample Questions

Separate file

Zombies

- **Idea**
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - 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 discards process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then 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

```
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 */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 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 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps** shows child process as “defunct”
- Killing parent allows child to be reaped by **init**

Orphan process: Nonterminating Child process

```
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);
    }
}
```

```
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
```

- Child process still active even though parent has terminated. The process init adopts the process. Daemons can be created this way.
- Must kill explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

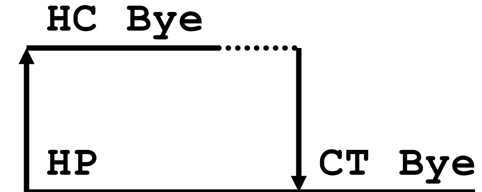
- **Parent reaps child by calling the `wait` function**

- **`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

`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()` Example

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

```
void fork10()
{
    pid_t 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++) {
        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 terminate abnormally\n", wpid);
    }
}
```

`waitpid()`: Waiting for a Specific Process

- `waitpid(pid, &status, options)`
  - suspends current process until specific process terminates
  - various options

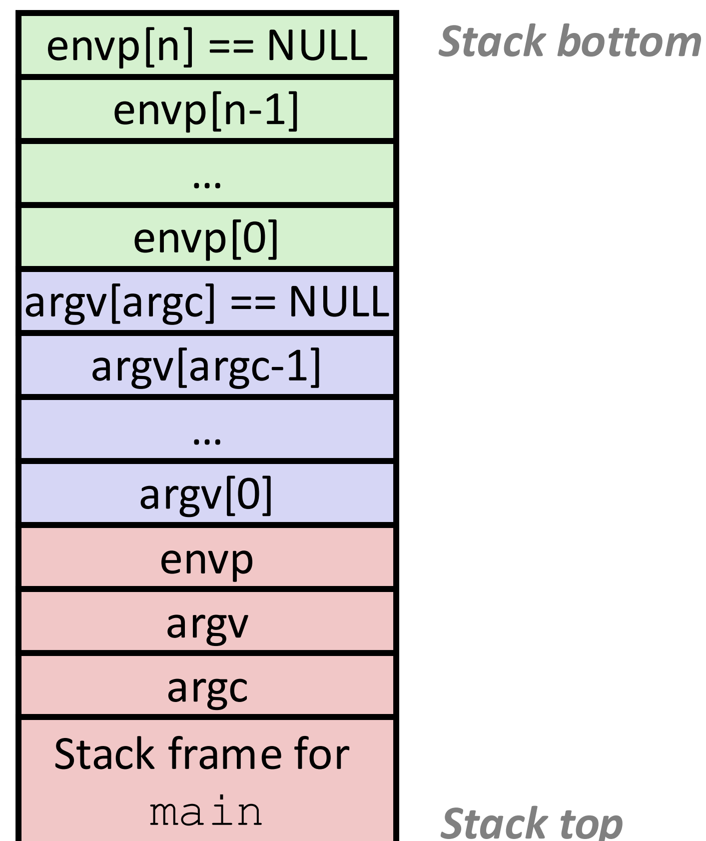
```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

execve: Loading and Running Programs

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

- **Loads and runs in current process:**
  - Executable **filename**
  - With argument list **argv**
  - And environment variable list **envp**
- **Does not return (unless error)**
- **Overwrites code, data, and stack**
  - keeps pid, open files
- **Environment variables:**
  - “name=value” strings
  - Use functions getenv and putenv to access environment variables.

The v and e comes from the fact that it takes an argument argv, envp to the vector of arguments and environment variables to the program



execve Example

```
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, envp) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```