Lecture 4: Processes (2)  
Threads
In Unix, processes are created using `fork()`

```
int fork()
```

- Creates and initializes a new PCB
- Creates a new address space
- Initializes the address space with a **copy** of the entire contents of the address space of the parent
- Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Places the PCB on the ready queue

Fork returns **twice**
- Returns the child’s PID to the parent, “0” to the child
int main(int argc, char *argv[]) {
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of \%s is \%d\n", name, getpid());
        return 0;
    } else {
        printf("My child is \%d\n", child_pid);
        return 0;
    }
}

What does this program print?
Example Output

[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

child_pid = 486

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}

child_pid = 0

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
Divergence

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Parent

```c
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Child

PC

child_pid = 486

child_pid = 0
Example Continued

[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
[well ~]$ ./a.out
Child of a.out is 498
My child is 498

Why is the output in a different order?
Why fork()?

- Very useful when the child...
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task

Example: Web server

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        // Handle client request
    } else {
        // Close socket
    }
}
```
Process Creation: Unix (2)

- Wait a second. How do we actually start a new program?
  
  ```c
  int exec(char *prog, char *argv[])
  ```

- `exec()`
  - Stops the current process
  - Loads the program “prog” into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

- What does it mean for `exec` to return?
- What does it mean for `exec` to return with an error?
Process Creation: Unix (3)

- fork() is used to create a new process, exec is used to load a program into the address space

- What happens if you run “exec csh” in your shell?
- What happens if you run “exec ls” in your shell? Try it.

- fork() can return an error. Why might this happen?
Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), NT: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note that a process does not need to clean up itself
  - OS will handle this on its behalf
wait() a second...

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use `wait()` (`WaitForSingleObject`)
  - Suspends the current process until a child process ends
  - `waitpid()` suspends until the specified child process ends
- Wait has a return value…what is it?
- Unix: Every process must be reaped by a parent
  - What happens if a parent process exits before a child?
  - What do you think is a “zombie” process?
Unix Shells

while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        if (!(run_in_background))
            waitpid(child_pid);
    }
}
Processes: check your understanding

- What are the units of execution?
  - Processes

- How are those units of execution represented?
  - Process Control Blocks (PCBs)

- How is work scheduled in the CPU?
  - Process states, process queues, context switches

- What are the possible execution states of a process?
  - Running, ready, waiting, ...

- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination

- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)
Processes

Recall that …
- A process includes:
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting info
  - Execution state (PC, SP, regs, etc.)
  - PCB to keep track of everything
- Processes are completely isolated from each other
Creating a new process is costly because of new address space and data structures that must be allocated and initialized
- Recall struct proc in xv6 or Solaris

Communicating between processes is costly because most communication goes through the OS
- Inter Process Communication (IPC) – we will discuss later
- Overhead of system calls and copying data
Parallel Programs

- Also recall our Web server example that forks off copies of itself to handle multiple simultaneous requests

- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
  - Have the OS schedule these processes in parallel

- This situation is very inefficient (CoW helps)
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)

- What don’t they share?
  - Each has its own execution state: PC, SP, and registers

- Key idea: Separate resources from execution state
- Exec state also called thread of control, or thread
Recap: Process Components

- A process is named using its process ID (PID)
- A process contains all of the state for a program in execution
  - An address space
  - The code for the executing program
  - The data for the executing program
  - A set of operating system resources
    - Open files, network connections, etc.
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - Current execution state (Ready/Running/Waiting)
Threads

- Separate execution and resource container roles
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space, resources, and general process attributes (everything but threads)

- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities
Recap: Process Address Space
Threads in a Process

Stack (T1)

Stack (T2)

Stack (T3)

Heap

Static Data

Code

Thread 1

Thread 2

Thread 3

PC (T2)

PC (T3)

PC (T1)
Thread Design Space

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Thread</th>
<th>One Thread/Process</th>
<th>Many Threads/Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Address Space</td>
<td>(MSDOS)</td>
<td></td>
<td>One Address Space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Pilot, Java)</td>
</tr>
<tr>
<td>Many Address Spaces</td>
<td>(Early Unix)</td>
<td></td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Mac OS, Unix, Windows)</td>
</tr>
</tbody>
</table>
Process/Thread Separation

- Separating threads and processes makes it easier to support multithreaded applications
  - Concurrency does not require creating new processes

- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events (e.g., Web requests)
  - Writing parallel programs

- So multithreading is even useful on a uniprocessor
Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task
- Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Threads: Concurrent Servers

Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

handle_request(int sock) {
    Process request
    close(sock);
}
```
Implementing threads

Kernel Level Threads

- All thread operations are implemented in the kernel
- The OS schedules all of the threads in the system
- Don’t have to separate from processes

OS-managed threads are called **kernel-level threads** or **lightweight processes**

- Windows: **threads**
- Solaris: **lightweight processes (LWP)**
- POSIX Threads (pthreads): **PTHREAD_SCOPE_SYSTEM**
Kernel Thread (KLT) Limitations

- KLTs make concurrency cheaper than processes
  - Much less state to allocate and initialize

- However, there are a couple of issues
  - Issue 1: KLT overhead still high
    » Thread operations still require system calls
    » Ideally, want thread operations to be as fast as a procedure call
  - Issue 2: KLTs are general; unaware of application needs

- Alternative: User-level threads (ULT)
Alternative: User-Level Threads

- Implement threads using user-level library

- ULTs are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    » No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS