EECS3221 notes

Jerry Wu

Week III 2023/05/23

Process management

Complete tuesday stuff later!

to complete

Memory layout of a C program

```
1 #include<stdio.h>
```

Multitasking

We want to be able to do multiple things at the same time on a computer system. This is where multitasking comes into play. In case one process crashes, we can use **multithreading** so that the whole system doesn't crash.

Process creation

- **Parent** processes create **child** processes which create other processes, forming a **process tree**.
- A process is identified and managed via a **process identifier (pid)**.
- Resource sharing options::
  - Parent and children share all resources
  - children share a subset of parents' resources
  - parent and child share no resources
- Execution options::

  - Parent and child processes execute concurrently
  - Parent process waits until child process terminates before terminating
  - Usually, killing a parent will also kill their children (wow, that's dark)
- Addresses space
  - Child can be a duplicate of parent
  - Child has a program loaded into it
- UNIX examples::
  - `fork()` is a system call that creates a new process
  - `exec()` is a system call used after fork to replace the process' memory space with a new program (there are multiple variations of `exec()`, read sytstem call API for more info)
  - Parent process calls `wait()` for the child to terminate
  - Always call `exit()` at the end of the program. This is done automatically by the OS, but it is good practice to put it at the end of your code.

Creating a process by forking

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    //fork a child process
    pid = fork();
    if(pid < 0) //error occurred because fork returned n<0
    {
        fprintf(stderr, "Fork failed");
        return 1;
    }

    else if(pid == 0) //child process
    {
        execlp("/bin/ls","ls",NULL);
    }
    else  //parent process
    {
    //parent will wait for the child to complete
        wait(NULL);
        printf("Child complete");
    }

    return 0;
}
```

Some things to note

- We use `fprintf()` in the error handling because we can **treat memory blocks as files in C**. Network output streams also behave in a similar manner and can be treated as files among other things as well.
- When the return value of `fork()` < 0, we have an error.
- When $return($`fork()`$) = 0$, we have the pid of the child process.
- `sys/types.h` is the system call library.

Process termination (good for dark humour)

- When calling `exit()`, the following things happen::
  - Returns status data from child to parent via `wait()`
  - Process' resources are deallocated by the OS
- Parent may terminate the execution of child processes using the `abort()` system call. Some reasons for doing so::
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue to run
  - Don't try the above as a person!
- Some OS do not allow child to exist if its parent has been terminated. If a process terminates, then all its children must also be terminated.
  - **Cascading termination** is what this is known as.
  - This is initiated by the OS
- The parent process may wait for termination of a child process by using `wait()` system call. The call returns status information and the pid of the terminated process. (`pid = wait(&status);`)
- If no parent is waiting (did not invoke `wait()`), the child is a **zombie**
- If the parent is terminated without invoking `wait()`, the child process is an **orphan**
- Moral of the story:: Always `wait` for your child(ren)!

Android process importance hierarchy

- Mobile OSs often have to terminate processes to reclaim system resources such as memory. They are terminated from least to most important:
  - Foreground (UI, MainActivity)
  - Visible processes (static text, images, etc.)
  - Services
  - Background processes
  - Empty processes
- Start down here

Interprocess communication

- Processes within a system may be **independent** or **cooperative**
- Cooperating processes can affect or be affected by other processes
- Reasons for cooperation::
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Producer consumer problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process