3: Processes

Programs vs Processes
- A program is passive
  - Sequence of commands waiting to be run
- A process is active
  - An instance of program being executed
  - There may be many processes running the same program
  - Also called job or task

What makes up a process?
- Address space
- Code
- Data
- Stack (nesting of procedure calls made)
- Register values (including the PC)
- Resources allocated to the process
  - Memory, open files, network connections

Address Space Map
- Stack (Space for local variables etc. for each nested procedure call)
- Heap (Space for memory dynamically allocated e.g. with malloc)
- Statically declared variables (global variables)
- Code (Text Segment)
- Stack Pointer
- PC

What kinds of processes are there?
- Compute bound/ IO bound
- Long-running/short-running
- Interactive/batch
- Large/small memory footprint
- Cooperating with other processes?
- ...
- How does the OS categorize processes?

Process States
- During their lifetime, processes move between various states
  - Ready - waiting for a turn to use the CPU
  - Running - currently executing on the CPU
  - How many processes can be in this state?
  - Waiting - Unable to use the CPU because blocked waiting for an event
  - Terminated/Zombie - Finished executing but state maintained until parent process retrieves state
State Transitions

New → Ready → Running → Terminated

State Queues

- OSes often maintain a number of queues of processes that represent the state of the processes
  - All the runnable processes are linked together into one queue
  - All the processes blocked (or perhaps blocked for a particular class of event) are linked together
  - As a process changes state, it is unlinked from one queue and linked into another

Context Switch

- When a process is running, some of its state is stored directly in the CPU (register values, etc.)
- When the OS stops a process, it must save all of this hardware state somewhere (PCB) so that it can be restored again
- The act of saving one processes hardware state and restoring another's is called a context switch
  - 100s or 1000s per second!

Schedulers

- Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU.

Schedulers (cont)

- Short-term scheduler is invoked very frequently (milliseconds) \(\Rightarrow\) (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) \(\Rightarrow\) (may be slow).
- The long-term scheduler controls the degree of multiprogramming.
- Processes can be described as either:
  - I/O-bound process - spends more time doing I/O than computations; many short CPU bursts.
  - CPU-bound process - spends more time doing computations; few very long CPU bursts.
Family Tree

- Age old questions - where do new processes come from?
- New processes are created when an existing process requests it
  - Creating process called the parent; created called the child
  - Children of some parent called siblings
  - Children often inherit privileges/attributes from their parent
  - Working directory, Clone of address space
  - When child is created, parent may either wait for it or continue in parallel

Init process

- In last stage of boot process, kernel creates a user level process, init
- Init is the parent (or grandparent…) of all other processes
- Init does various important housecleaning activities
  - checks and mounts the filesystems, sets hostname, timezones, etc.
- Init reads various "resource configuration files" (/etc/rc.conf, etc) and spawns off processes to provide various services
- In multi-user mode, init maintains processes for each terminal port (tty)
  - Usually runs getty which executes the login program

Management of PCBs

- PCBs are data structures (just like you are used to at user level)
- Space for them may be dynamically allocated as needed or perhaps a fixed sized array of PCBs for the maximum number of possible processes is allocated at init time
- As process is created, a PCB is assigned and initialized for it
- Often process id is an offset into an array of PCBs
- After process terminates, PCB is freed (sometimes kept around for parent to retrieve its exit status)
State Queues

Tail ptr
Head ptr
Prev next
Prev next
Prev next
Rest of PCB
Rest of PCB
Rest of PCB

Ready queue, queues per device, queue of all processes, ...

Context Switch
- When a process is running, some of its state is stored directly in the CPU (register values, etc.)
- When the OS stops a process, it must save all of this hardware state somewhere (PCB) so that it can be restored again
- The act of saving one process's hardware state and restoring another's is called a context switch
  - 100s or 1000s per second!

UNIX process creation
- `fork()` system call
  - Creates a new PCB and a new address space
  - Initializes the new address space with a *copy* of the parent's address space
  - Initializes many other resources to copies of the parents (e.g. same open files)
  - Places new process on the queue of runnable processes
- `fork()` returns twice: to parent and child
  - Returns child's process ID to the parent
  - Returns 0 to the child

Example Code Snippet
```c
int main (int argc, char **argv)
{
    int childPid;
    childPid = fork();
    if (childPid == 0) {
        printf("Child running\n");
    } else {
        printf("Parent running: my child is %d\n", childPid);
    }
}
```

Output
```bash
% ./tryfork
Parent running: my child is 707
Child running
%
```

Experiments
- Try putting an infinite loop in the child's portion (do you return to the command shell?) and then looking for it in the `ps` output
- Try putting an infinite loop in the parent's portion (do you return to the command shell?)
- Put an infinite loop in both
  - try killing the child (look in the `ps` output for the child and the parent)
  - Try killing the parent - what happens to the child?
- WARNING: DO NOT PUT THE FORK COMMAND ITSELF IN AN INFINITE LOOP!!! YOU WILL CRASH THE SYSTEM!
Fork and Exec

- How do we get a brand new process not just a copy of the parent?
  - Exec() system call
  - int exec (char * prog, char ** argv)

Exec:
- Stops the current process
- Loads the program, prog, into the address space
- Passes the arguments specified in argv
- Places the PCB back on the ready queue

Exec "takes over" the process:
- There is no going back to it when it returns
- Try to exec something in your shell (example: exec ls) – when ls is done your shell is gone because is replaced!

UNIX Shell

```c
int main (int argc , char ** argv)
{
    // code...
}
```

Windows Process Creation

```c
BOOL CreateProcess( 
    LPCTSTR lpApplicationName, // name of executable
    module LPCTSTR *lpCommandLine, // command line string
    LPSECURITY_ATTRIBUTES*lpProcessAttributes, // SD
    LPSECURITY_ATTRIBUTES*lpThreadAttributes, // SD
    BOOL bInheritHandles, // handle inheritance option
    DWORD dwCreationFlags, // creation flags
    LPVOID lpEnvironment, // new environment block
    LPCTSTR lpCurrentDirectory, // current directory name
    LPSTARTUPINFO lpStartupInfo, // startup information
    LPPROCESS_INFORMATION lpProcessInformation // process information
);```

Windows vs Unix

- Windows doesn't maintain the same relationship between parent and child
- Later versions of Windows have concept of “job” to mirror UNIX notion of parent and children (process groups)
- Waiting for a process to complete?
  - WaitForSingleObject to wait for completion
  - GetExitCodeProcess (will return STILL_ALIVE until process has terminated)

Cooperating Processes

- Processes can run independently of each other or processes can coordinate their activities with other processes
- To cooperate, processes must use OS facilities to communicate
  - Many others
    - Shared Memory
    - File
    - Sockets
    - Pipes
    - Signals
    - Events
    - Remote Procedure Call

Sockets

- A socket is an end-point for communication over the network
- Create a socket
  - int socket (int domain, int type, int protocol)
  - Type = SOCK_STREAM for TCP
- Read and write socket just like files
  - Can be used for communication between two processes on same machine or over the network
**Pipes**
- Bi-directional data channel between two processes on the same machine
- Created with:
  - `int pipe (int fildes[2])`
- Read and write like files

**Signals**
- Processes can register to handle signals with the `signal` function
  - `void signal (int signum, void (*proc) (int))`
- Processes can send signals with the `kill` function
  - `kill (pid, signum)`
- System defined signals like `SIGHUP (0), SIGKILL (9), SIGSEGV(11)`
  - In UNIX shell, try: `kill -9 pidOfProcessYouDon'tReallyCareAbout`
- Signals not used by system like `SIGUSR1` and `SIGUSR2`
- Note: `sigsend/sigaction` similar to `kill/signal`

**Remote Procedure Call (RPC)**

**Processes**
- What is a process?
- Process States
- Switching Between Processes
- Process Creation
- PCBs
- Communication/Cooperation between processes

**Outtakes**
- Could spend more time on things in Process Creation and Signal chapter of Stevens