Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process is the unit of work within the system.
- Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process.
- Systems consist of a collection of processes: operating-system processes and user processes.
- The CPU switch from one to another (multiprogramming)
  - Recall that only one program active at any instant of time.

Process Concept (cont’d)

- A process executes a program, but a program is not a process.
  - A program is passive (a set of instructions).
  - A process is active (a program in execution).
  - Multiple processes may execute the same program
- A process contains all of the state for a program in execution
  - An address space
  - The code (called text sometimes) for the executing program
  - The data for the executing program
  - 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
  - A set of operating system resources
    - Open files, network connections, etc.
- A process may contain a Heap, which is a memory that is allocated during process run-time.
- A process is named using its process ID (PID)
A process has an execution state that indicates what it is currently doing.

- As a process executes, it changes state:
  - **new**: The process is being created
  - **running**: Instructions are being executed on a CPU
  - **waiting**: The process is waiting for some event to occur
    - It cannot make progress until event is signaled (disk completes)
  - **ready**: The process is waiting to be assigned to a CPU
    - Ready to execute, but another process is executing on the CPU
  - **terminated**: The process has finished execution

- Unix command “ps”: STAT column indicates execution state
### Process Control Block (PCB)

- Each process is represented in the OS by a PCB
- PCB includes information associated with each process:
  - Process state: any of the above states.
  - Program counter: next instruction to be executed
  - CPU registers: must be saved during state changes
  - CPU scheduling information such as priority
  - Memory-management information: base register, page tables, segment tables
  - Accounting information such as CPU cycles or time.
  - I/O status information: allocated devices, open files
- The PCB is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware
PCBs and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
- The process of changing the CPU hardware state from one process to another is called a context switch
  - This can happen 100 or 1000 times a second!
Context Switch

- When the CPU is switched to another process, the system must save the state of the old process and load the previously saved state for the new process. This is called **context switching**.
- The context is represented in the PCB of the process:
  - Value of CPU registers
  - Process state
  - Memory-management information
- Context-switch time is an overhead; the CPU doesn't do useful work while switching between processes, so it should be done fast.
- Time dependent on hardware support:
  - Memory speed
  - Number of registers
  - Special instructions
Process Scheduling

- Multiprogramming: the goal is to have some process running at all time to maximize the CPU utilization
- Time-sharing: the goal is to switch the CPU among processes so frequently that users can interact with each program while it is running
- Done by process scheduler.
  - Note that for single-processor system, there will never be more than one running process at the same time.

Process Scheduling Queues

How does the OS keep track of processes?
- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has one queue for each state
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another
- Examples:
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
    - Generally stored as linked list of PCB's
  - Device queues – set of processes waiting for an I/O device
    - Each device has its own device queue
Process Queues

- Ready Queue
- Netscape PCB
- X Server PCB
- Idle PCB
- Disk I/O Queue
- Emacs PCB
- io PCB
- Console Queue
- Sleep Queue

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.).

Representation of Process Scheduling

- New processes in ready queue
- Wait until selected for execution (dispatched)
- Then,
  - I/O request \( \rightarrow \) I/O queue
  - Create new sub process \( \rightarrow \) wait for finishing
  - Removed forcibly from CPU by interrupt \( \rightarrow \) back to ready queue
- After waiting, the process eventually becomes ready and moved to the ready queue. This process continues until the process terminates.
Schedulers

- A process moves among various queues. The OS must select processes from these queues. This is carried out by the appropriate scheduler.
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue from a mass-storage device (typically, a disk)
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU to it
- **Medium-term scheduler** may be used in time-sharing systems with swapping. (remove process from memory to decrease degree of multiprogramming, then restore the process later)

Schedulers (cont’d)

- Frequency of execution is the main distinction among different types of schedulers.
- Short-term scheduler is invoked frequently.
  - (milliseconds) ➔ (must be fast)
- Long-term scheduler is invoked infrequently.
  - (seconds, minutes) ➔ (may be slower)
  - The long-term scheduler controls the degree of multiprogramming. (*Number of processes in memory*)
  - It is important for the long-term scheduler to select a good process mix.
- A process 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.
Addition of Medium Term Scheduling

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
  - (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 0 or 1))
  - Typically, a new process is created by having an existing process executes a process creation system call
- Processes are identified by a unique process identifier (PID)
- Resource sharing
  - Parents and children share all resources
  - Children share subset of parent’s resources
  - Ex: Unix: Process User ID is inherited – children of your shell execute with your privileges
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminates
Process Creation (Cont.)

- Address space
  - Child duplicate of parent (same program code and date)
  - Child has a new program loaded into it
- UNIX examples
  - **Fork** system call creates new process
    - same address space i.e. an exact clone of the calling process (easy communication between them)
    - Executes concurrently
    - Return value from **fork** distinguishes which process is which
      - Returns the child's PID to the parent, "0" to the child
  - **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
- Note: after a process is created, both the parent and child have their own distinct address spaces

---

Process Creation

![Diagram](image)

- fork()
  - parent
  - child
- wait
  - exec()
  - exit()
Process Creation : NT

- The system call on NT for creating a process is called, surprisingly enough, CreateProcess:
- **BOOL CreateProcess(char *prog, char *args) (simplified)**
- CreateProcess
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - **Loads** the program specified by “prog” into the address space
  - Copies “args” into memory allocated in address space
  - Initializes the hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the ready queue

Fork Example

```c
int main(int argc, char *argv[])
{
    char *name = argv[0];
    int return_pid = fork();
    if (return_pid == 0) {
        printf("Child of %s has PID %d \n", name, getpid());
        return 0;
    } else {
        printf("I'm the parent, My child PID=%d \ My PID=%d", return_pid, getpid());
        return 0;
    }
}
```

What does this program print?
**Fork Example (contd)**

sunv880% gcc fork1.c  
sunv880% a.out  
I'm the Parent, My Child PID=24369      My PID =24368  
Child of a.out has PID=24369

**Duplicating Address Spaces**

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

Parent

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

Child
Divergence

Example (Contd)

sunv880% a.out
I'm the Parent, My Child PID=24386   My PID =24385
Child of a.out has PID=24386

sunv880% a.out
Child of a.out has PID=24389
I'm the Parent, My Child PID=24389   My PID =24388

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 (contd)**

- How do we actually start a new program?
  - Usually `exec()` system call used after a `fork` to replace the process' memory space with a new program (destroying the memory image of the program containing the `exec` system call
  ```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 (contd)

- 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?

Another Example: C Program Forking Separate Process

```c
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-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");
        exit(0);
    }
}
```
**Process Termination**

- **Conditions which terminate processes**
  - Normal exit (voluntary) → finish job.
  - Error exit (voluntary) → after compilation errors, for example.
  - Fatal error (involuntary) → illegal instruction, illegal memory reference
  - Killed by another process (involuntary) → Ctrl+Alt+Del

- **Process executes last statement and asks the operating system to delete it voluntarily (exit)**
  - Output data from child to parent (via wait)
  - Process’ resources are deallocated by operating system

- **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
Process Termination (Contd)

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates (not the case in either Windows or UNIX)
      - All children terminated - cascading termination
- 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

Process Summary

- 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?
  - New, Running, ready, waiting, Terminated
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)
End of Chapter 3