Chapter 3
Processes

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017
Outline

- Concept of process
- (UNIX-centric) User view of processes
- Kernel view of processes
- Interprocess Communication
What is a “process”? 

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources
Process in Memory

![Diagram showing memory layout with sections for stack, heap, data, and text.](image)
Process Elements

- A process is comprised of:
  - Program code (possibly shared)
  - A set of data
  - A number of attributes describing the state of the process
Trace of the Process

The behavior of an individual process is shown by listing the sequence of instructions that are executed.

This list is called a *Trace*.

*Dispatcher* is a small program which switches the processor from one process to another.
Consider three processes being executed
All are in memory (plus the dispatcher)
Let's ignore virtual memory for this.
Trace from the *processes* point of view:

- Each process runs to completion

```
5000 8000 12000

(a) Trace of Process A  (b) Trace of Process B  (c) Trace of Process C
```

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2
Trace from Processors point of view

100 = Starting address of dispatcher program
Shaded areas indicate execution of dispatcher process; first and third columns count instruction cycles; second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2
Outline

- Concept of process
- (UNIX-centric) User view of processes
- Kernel view of processes
- Interprocess Communication
Process Creation

- The OS builds a data structure to manage the process

- Traditionally, the OS created all processes
  - But it can be useful to let a running process create another

- This action is called *process spawning*
  - *Parent Process* is the original, creating, process
  - *Child Process* is the new process
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
Processes Tree on Solaris

- **Sched**
  - pid = 0
- **init**
  - pid = 1
  - **inetd**
    - pid = 140
  - **telnetdaemon**
    - pid = 7776
    - **Csh**
      - pid = 7778
      - **Netscape**
        - pid = 7785
      - **emacs**
        - pid = 8105
  - **pageout**
    - pid = 2
  - **fsflush**
    - pid = 3
  - **dtlogin**
    - pid = 251
    - **Xsession**
      - pid = 294
    - **sdt_shel**
      - pid = 340
    - **Csh**
      - pid = 1400
      - **ls**
        - pid = 2123
      - **cat**
        - pid = 2536
Process Creation (Cont.)

- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.

- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program.
Process Creation (UNIX)

- fork()
- child
- exec()
- parent
- wait
- exit()
- resumes
pid = fork();
if (pid < 0){/*error occurred*/
    fprintf(stderr, “Fork failed”);
    exit(-1);}
else if(pid == 0){/*child process*/
    execvp(“/bin/ls”,”ls”,NULL);
}
else { /*parent process*/
    wait(NULL);
    printf(“Child Complete”);
    exit(0);
}
Process Termination

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

- Parent may terminate execution of children processes (abort).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.

  ✓ Operating system does not allow child to continue if its parent terminates.

  ✓ Cascading termination.
Creating processes

- int fork (void);
  - Create new process that is exact copy of current one
  - Returns process ID of new process in “parent”
  - Returns 0 in “child”

- int waitpid (int pid, int *stat, int opt);
  - pid – process to wait for, or -1 for any
  - stat – will contain exit value, or signal
  - opt – usually 0 or WNOHANG
  - Returns process ID or -1 on error
Deleting processes

- **void exit (int status);**
  - Current process ceases to exist
  - status shows up in waitpid (shifted)
  - By convention, status of 0 is success, non-zero error

- **int kill (int pid, int sig);**
  - Sends signal sig to process pid
  - SIGTERM most common value, kills process by default
  - (but application can catch it for “cleanup”)
  - SIGKILL stronger, kills process always
Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to main
  - `envp` – environment variables, e.g., PATH, HOME

- Generally called through a wrapper functions
  - `int execvp (char *prog, char **argv);`
    - Search PATH for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    - List arguments one at a time, finish with NULL

- Example
Outline

- Concept of process
- (UNIX-centric) User view of processes
- Kernel view of processes
- Interprocess Communication
Recall: Traditional UNIX Process

- Process: Operating system abstraction to represent what is needed to run a single program
  - Often called a “HeavyWeight Process”

- Two parts:
  - Sequential program execution stream
    - Code executed as a sequential stream of execution (i.e., thread)
    - Includes State of CPU registers
  - Protected resources:
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)
Process Elements

While the process is running it has a number of elements including:

- Identifier
- State
- Priority
- Program counter
- Memory pointers
- Context data
- I/O status information
- Accounting information
Implementing processes

- Keep a data structure for each process
  - Process Control Block (PCB)
  - Called proc in Unix, task_struct in Linux,
- Tracks state of the process
  - Running, ready (runnable), waiting, etc.
- Includes information necessary to run
  - Registers, virtual memory mappings, etc.
  - Open files, including memory mapped files
- Various other data about the process
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, . . .

<table>
<thead>
<tr>
<th>Process state</th>
<th>PCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID</td>
<td></td>
</tr>
<tr>
<td>User id, etc.</td>
<td></td>
</tr>
<tr>
<td>Program counter</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
</tr>
<tr>
<td>Address space</td>
<td></td>
</tr>
<tr>
<td>(VM data structs)</td>
<td></td>
</tr>
<tr>
<td>Open files</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How do we Multiplex Processes?

- The current state of process held in a process control block (PCB):
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time

- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes

- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Example mechanisms:
    - Memory Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls
Process Control Block (PCB)

- Pointer
- Process state
- Process number
- Program counter
- Registers
- Memory limits
- List of open files
- ..
- ..
Process State

- As a process executes, it changes *state*
  - **new**: The process is being created.
  - **ready**: The process is waiting to be assigned to a processor.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur.
  - **terminated**: The process has finished execution.
- Process can be in one of several states
  - new & terminated at beginning & end of life
  - running – currently executing (or will execute on kernel return)
  - ready – can run, but kernel has chosen different process to run
  - waiting – needs async event (e.g., disk operation) to proceed

- Which process should kernel run?
  - if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
  - if >1 runnable, must make scheduling decision
Diagram of Process State

New → Admitted → Ready

Ready → Running → Exit

Running → Waiting → Scheduler Dispatch

Waiting → I/O or Event Completion → Ready

I/O or Event Completion → I/O or Event Wait → Waiting
Process Creation

- Once the OS decides to create a new process it:
  - Assigns a unique process identifier
  - Allocates space for the process
  - Initializes process control block
  - Sets up appropriate linkages
  - Creates or expand other data structures
When to switch processes

A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Cause</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>External to the execution of the current instruction</td>
<td>Reaction to an asynchronous external event</td>
</tr>
<tr>
<td>Trap</td>
<td>Associated with the execution of the current instruction</td>
<td>Handling of an error or an exception condition</td>
</tr>
<tr>
<td>Supervisor call</td>
<td>Explicit request</td>
<td>Call to an operating system function</td>
</tr>
</tbody>
</table>

Table 3.8 Mechanisms for Interrupting the Execution of a Process
CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
The steps in a process switch are:

1. Save context of processor including program counter and other registers
2. Update the process control block of the process that is currently in the Running state
3. Move process control block to appropriate queue
Change of Process State cont…

4. Select another process for execution
5. Update the process control block of the process selected
6. Update memory-management data structures
7. Restore context of the selected process
Process Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** – set of processes waiting for an I/O device.
- **Process migration** between the various queues.
Ready Queue And Various I/O Device Queues

- Ready Queue
- Mag Tape Unit 0
- Mag Tape Unit 1
- Disk Unit 0
- Terminal Unit 0

Diagram showing queue headers and PCBs.
PCBs move from queue to queue as they change state

- Decisions about which order to remove from queues are Scheduling decisions
- Many algorithms possible (few weeks from now)
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be loaded into memory for execution.

- **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) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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.
Addition of Medium-Term Scheduling

- swap in
- partially executed swapped-out processes
- swap out
- ready queue
- CPU
- I/O
- I/O waiting queues
- end
Outline

- Concept of process
- (UNIX-centric) User view of processes
- Kernel view of processes
- Interprocess Communication
Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process.

Advantages of process cooperation:

- Information sharing
- Computation speed-up
- Modularity
- Convenience
Communication Models

- Communication may take place using either message passing or shared memory.
Communication Models

- Message passing
  - Smaller data exchange
  - Intercomputer communication
  - System call with kernel intervention

- Shared memory
  - Maximum speed/memory
  - Convenience of communication
  - Protection and synchronization
  - Routine memory access without kernel intervention
Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
Bounded-Buffer – Shared-Memory Solution

- Shared data

```c
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer process

```c
item nextProduced;

while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer process

```c
item nextConsumed;

while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```
Bounded-Buffer

- Solution is correct, but can only use BUFFER_SIZE-1 elements
POSIX Shared Memory Example
(see textbook page 90)

- **shmget()**: A process creates a shared memory segment using this function.

- **shmct1()**: the original owner of a shared memory segment can assign ownership to another user with this function.

- **shmat()**: Once created, a shared segment can be attached to a process address space using this.

- **shmdt()**: shared segment can be detached using this.
Shared Memory Example

- http://www.cs.cf.ac.uk/Dave/C/node27.html
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
Interprocess Communication (Cont.)

- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`

- If $P$ and $Q$ wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via `send/receive`

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
Direct Communication

Processes must name each other explicitly:

- **send** \( (P, \text{message}) \) – send a message to process \( P \)
- **receive** \( (Q, \text{message}) \) – receive a message from process \( Q \)

Properties of communication link

- Links are established automatically.
- A link is associated with exactly one pair of communicating processes.
- Between each pair there exists exactly one link.
- The link may be unidirectional, but is usually bidirectional.
Indirect Communication

Messages are directed and received from mailboxes (also referred to as ports).
- Each mailbox has a unique id.
- can communicate only if they share a mailbox.

Properties of communication link
- Link established only if processes share a common mailbox
- A link may be associated with many processes.
- Each pair of processes may share several communication links.
- Link may be unidirectional or bi-directional.
Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - `send(A, message)` – send a message to mailbox A
  - `receive(A, message)` – receive a message from mailbox A
Indirect Communication

- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A.
  - $P_1$, sends; $P_2$ and $P_3$ receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.
Buffering

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of $n$ messages
     Sender must wait if link full.
  3. Unbounded capacity – infinite length
     Sender never blocks.
Pipe

- UNIX pipes are implemented in a similar way, but with the pipe() system call.
  - The output of one process is connected to an in-kernel pipe.
  - The input of another process is connected to that same pipe.
  - E.g.,
    - `ls | wc`
```c
if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
}
else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);

    /* close the write end of the pipe */
    close(fd[READ_END]);
```
Pipe Example

- Advanced Linux Programming: Page 110
- When you invoke the command `ls | wc`, two forks occur: one for the `ls` child process and one for the `less` child process. Both of these processes inherit the pipe file descriptors so they can communicate using a pipe.
Comparison between different Interprocess Communication methods

- **Shared Memory**: permits processes to communicate by simply reading and writing to a specified memory location.
- **Pipe**: permit sequential communication from one process to a related process.
- **Sockets**: support communication between unrelated processes even on different computers.