CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2019 Lecture 5
Processes

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
• *Memory* is allocated for the kernel and the user processes. Each user process has its own address space.

• **Program control Block:** all the info needed to restart a process: process id, process state, registers, info about memory, open files etc.

• **Where is PCB saved?** memory area that is protected from user access perhaps as structs in a linked list.

• **What happens to a process once it is finished?** Resources deallocated, but only after..

• If a process forks a child, what happens to the parent? It continues.

• When the CPU is running user-processes how does the OS run?
• Can multiple processes/threads run on a single CPU? Yes, “concurrently” but not in “parallel” without additional hardware.

• Why fork? The only way to create new processes, except for init.

• BSD (Berkeley Software Distribution) is a derivative of original Unix.

• Linux is an implementation of Unix. Clone. “Unix-like”

• Process Scheduling – How? more soon
Diagram of Process State

Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue

Running to Waiting (Blocked): process blocks for input/output
Waiting to Ready: Input available
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources*
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
A Tree of Processes in Linux
• 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
Fork ( ) to create a child process

- Fork creates a copy of process
- Return value from fork (): integer
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Perhaps exceeds resource constraints. sets errno (a global variable in errno.h)
    - Running in original process

- All of the state of original process duplicated in both Parent and Child! Almost..
  - Memory, File Descriptors (next topic), etc...
Process Management System Calls

• UNIX fork – system call to create a copy of the current process, and start it running
  — No arguments!
• UNIX exec – system call to change the program being run by the current process. Several variations.
• UNIX wait – system call to wait for a process to finish
• Details: see man pages

Some examples:

```c
pid_t pid = getpid(); /* get current processes PID */;
waitpid(cid, 0, 0); /* Wait for my child to terminate. */
exit (0); /* Quit*/
kill(cid, SIGKILL); /* Kill child*/
```
UNIX Process Management

```
fork

pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);

child

exec

main () {
  ...
}

wait

parent
```
C Program Forking Separate Process

```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 */
    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;
}
```

<sys/types.h> definitions of derived types
<unistd.h> POSIX API

execlp(3) - Linux man page
http://linux.die.net/man/3/execlp
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t cid;

    /* fork a child process */
    cid = fork();
    if (cid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (cid == 0) { /* child process */
        printf("I am the child %d, my PID is %d\n", cid, getpid());
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent with PID %d, my parent is %d, my child is %d\n", getppid(), getpid(), cid);
        wait(NULL);

        printf("Child Complete\n");
    }

    return 0;
}
• Wait/waitpid ( ) allows caller to suspend execution until child’s status is available
• Process status availability
  – Generally after termination
  – Or if process is stopped
• `pid_t waitpid(pid_t pid, int *status, int options);`
• The value of pid can be:
  – 0 wait for any child process with same `process group ID` (perhaps inherited)
  – >0 wait for child whose process group ID is equal to the value of pid
  – -1 wait for any child process – (equi to wait ( ))
• Status: where status info needs to be saved
• Search for man fork( )
• http://man7.org/linux/man-pages/man2/fork.2.html

NAME  fork - create a child process
SYNOPSIS  #include <unistd.h>
            pid_t fork(void);

DESCRIPTION  fork() creates a new process by duplicating the calling
             process. The new process is referred to as the child process. …
             The child process and the parent process run in separate memory spaces…
             The child process is an exact duplicate of the parent process except
             for the following points: …. 

RETURN VALUE  On success, the PID of the child process is returned in the
             parent, and 0 is returned in the child. On failure, -1 is returned in the
             parent, no child process is created, and errno is set appropriately.
EXAMPLE  See pipe(2) and wait(2).
…

errno is a global variable in errno.h
Process Group ID

- Process group is a collection of processes
- Each process has a process group ID
- Process group leader?
  - Process with pid==pgid
- kill treats negative pid as pgid
  - Sends signal to all constituent processes
- A child Inherits parent’s process group ID
  - Parent can change group ID of child by using setpgid
  - Child can give itself new process group ID
Process Groups

By default, comprises:

• Parent (and further ancestors)
• Siblings
• Children (and further descendants)

A process can only send signals to members of its process group

• Signals are a limited form of inter-process communication used in Unix.
Process Termination

• Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  – Returns status data from child to parent (via `wait()`)
  – Process’ resources are deallocated by operating system

• Parent may terminate the execution of children processes using the `kill()` 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 if its parent terminates

```c
kill(child_pid,SIGKILL);
```
Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.

- **cascading termination.** All children, grandchildren, etc. are terminated.
- The termination is initiated by the operating system.

The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process:

```
pid = wait(&status);
```

If no parent waiting (did not invoke `wait()`) process is a **zombie**

If parent terminated without invoking `wait`, process is an orphan (it is still running, reclaimed by init)

---

**Zombie:** a process that has completed execution (via the exit system call) but still has an entry in the process table.
Meanwhile, on an ordinary Linux kernel...

What's going on with these zombie processes?

Their parent is too busy to get any notifications...

Daniel Storie {turnoff.us}
Multi-process Program Ex – Chrome Browser

• Early web browsers ran as single process
  – If one web site causes trouble, entire browser can hang or crash

• Google Chrome Browser is multiprocess with 3 different types of processes:
  – **Browser** process manages user interface, disk and network I/O
  – **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    • Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  – **Plug-in** process for each type of plug-in
Multitasking
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
Processes within a system may be **independent** or **cooperating**

Cooperating process can affect or be affected by other processes, including sharing data

Reasons for cooperating processes:
- Information sharing
- Computation speedup
- Modularity
- Convenience

Cooperating processes need **interprocess communication (IPC)**

Two models of IPC
- **Shared memory**
- **Message passing**
(a) Message passing. (b) shared memory.
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;
  ```

- in points to the next free position in the buffer
- out points to the first full position in the buffer.
- Buffer is empty when `in == out`;
- Buffer is full when `((in + 1) % BUFFER_SIZE) == out`. (Circular buffer)
- This scheme can only use `BUFFER_SIZE-1` elements

![Diagram of bounded buffer](image-url)
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```c
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```
Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate.
- The communication is under the control of the users processes, not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  - Synchronization is discussed in great details in Chapter 6.
- Example soon.

Only one process may access shared memory at a time.
• Mechanism for processes to communicate and to synchronize their actions

• Message system – processes communicate with each other without resorting to shared variables

• IPC facility provides two operations:
  – send(message)
  – receive(message)

• The message size is either fixed or variable
• If processes $P$ and $Q$ wish to communicate, they need to:
  – Establish a *communication link* between them
  – Exchange messages via send/receive
• Implementation issues:
  – 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?
• Implementation of communication link
  – Physical:
    • Shared memory
    • Hardware bus
    • Network
  – Logical: Options (details next)
    • Direct (process to process) or indirect (mail box)
    • Synchronous (blocking) or asynchronous (non-blocking)
    • Automatic or explicit buffering
Direct Communication

• Processes must name each other explicitly:
  – \texttt{send} \((P, message)\) – send a message to process \(P\)
  – \texttt{receive} \((Q, message)\) – 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
  – Processes 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 (port)
  – send and receive messages through mailbox
  – destroy a mailbox

• Primitives are defined as:
  \texttt{send}(A, \textit{message}) – send a message to mailbox A
  \texttt{receive}(A, \textit{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?

• Possible 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.
Message passing may be either blocking or non-blocking

**Blocking** is termed **synchronous**
- **Blocking send** -- sender is blocked until message is received
- **Blocking receive** -- receiver is blocked until a message is available

**Non-blocking** is termed **asynchronous**
- **Non-blocking send** -- sender sends message and continues
- **Non-blocking receive** -- the receiver receives:
  - A valid message, or
  - Null message

Different combinations possible
- If both send and receive are blocking, we have a **rendezvous**.
- Producer-Consumer problem: Easy if both block
Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of \( n \) messages Sender must wait if queue full
  3. Unbounded capacity – infinite length Sender never waits
Examples of IPC Systems - POSIX

- Older scheme (System V) usingshmget(), shmat(), shmdt(), shmmctl()

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```c
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ```
    - Returns file descriptor (int) which identifies the file
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```c
    ftruncate(shm_fd, 4096);
    ```
  - map the shared memory segment in the address space of the process
    ```c
    ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    ```
  - Now the process could write to the shared memory
    ```c
    sprintf(ptr, "Writing to shared memory");
    ```
Examples of IPC Systems - POSIX

POSIX Shared Memory

- Other process opens shared memory object `name`
  ```c
  shm_fd = shm_open(name, O_RDONLY, 0666);
  ```
  - Returns file descriptor (int) which identifies the file
- map the shared memory object
  ```c
  ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  ```
- Now the process can read from to the shared memory object
  ```c
  printf("%s", (char *)ptr);
  ```
- remove the shared memory object
  ```c
  shm_unlink(name);
  ```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "DS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
/* create the shared memory segment */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

/* configure the size of the shared memory segment */
 ftruncate(shm_fd,SIZE);

/* now map the shared memory segment in the address space of the process */
ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    return -1;
}

/**
 * Now write to the shared memory region.
 *
 * Note we must increment the value of ptr after each write.
 */
 sprintf(ptr,"%s",message0);
ptr += strlen(message0);
 sprintf(ptr,"%s",message1);
ptr += strlen(message1);
 sprintf(ptr,"%s",message2);
ptr += strlen(message2);

return 0;
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "CS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
/* open the shared memory segment */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    if (shm_fd == -1) {
        printf("shared memory failed\n");
        exit(-1);
    }

    /* now map the shared memory segment in the address space of the process */
    ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        exit(-1);
    }

    /* now read from the shared memory region */
    printf("%s",ptr);

    /* remove the shared memory segment */
    if (shm_unlink(name) == -1) {
        printf("Error removing %s\n",name);
        exit(-1);
    }