CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Spring 2019 Lecture 5
Processes

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• Where does the parent process come from?

• Why fork? The only way to create new processes.

• Program can involve multiple processes that run concurrently perhaps even in parallel if the hardware is available.

• Each process has its own separate address space, unless..

• Why fork( ) in child returns a 0? So that a child can identify itself. It is not the PPID of the child.

• How does a child know who is its parent? getppid()

How does the parent know the child’s PID? Returned by fork(). See example
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 (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
UNIX Process Management

fork

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

child

exec

main () {
  ...
}

wait

parent

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

#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",getpid(), getppid(), 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
• 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);
```
Process Termination

• 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

\[ \text{pid} = \text{wait}(&\text{status}); \]
• 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 Stori {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
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 processes can affect or be affected by other processes, including sharing data. Reasons for cooperating processes include:
- Information sharing
- Computation speedup
- Modularity
- Convenience

Cooperating processes need interprocess communication (IPC). Two models of IPC are:
- 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;
  ```

  ```c
  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
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;
}
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 */
}
• 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 5.
• Example soon.
Interprocess Communication – Message Passing

• 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:
  – send \((P, message)\) – send a message to process P
  – 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 bi-directional
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, message) – send a message to mailbox A
  \texttt{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?

• 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.
Synchronization *(blocking or not)*

- 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) using `shmget()`, `shmat()`, `shmdt()`, `shmctl()`

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    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
    ```
    ftruncate(shm_fd, 4096);
    ```
  - 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);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(ptr, "Writing to shared memory");
    ```
Examples of IPC Systems - POSIX

**POSIX Shared Memory**

- Other process opens shared memory object
  
  ```
  shm_fd = shm_open(name, O_RDONLY, 0666);
  ```
  
  - Returns file descriptor (int) which identifies the file

- map the shared memory object

  ```
  ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  ```

- Now the process can read from the shared memory object

  ```
  printf("%s", (char *)ptr);
  ```

- remove the shared memory object

  ```
  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 = "OS";
    /* 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);
}
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)
Sockets

• A **socket** is defined as an endpoint for communication

• Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host

• The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

• Communication consists between a pair of sockets

• All ports below 1024 are **well known**, used for standard services

• Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running
Socket Communication

- **CS457** Computer Networks and the Internet
Pipes

- Acts as a conduit allowing two processes to communicate
- One of the first IPC implementation mechanisms
Pipes

• Conduit allowing two processes to communicate

• Issues:
  – Is communication unidirectional or bidirectional?
  – If bidirectional, is it half-duplex (one way at a time) or full-duplex (both directions simultaneously)?
  – Must there exist a relationship (i.e., parent-child) between the communicating processes?
  – Can the pipes be used over a network?
Pipes

- Command line:
  - Set up pipe between commands
    `ls | more`
    Output of `ls` delivered as input to `more`

- Ordinary ("anonymous") pipes – Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Cannot be accessed from outside the process that created it.

- Named pipes ("FIFO") – can be accessed without a parent-child relationship.
Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional (half duplex)
- **Require parent-child relationship** between communicating processes
- **pipe (int fd[])** to create pipe, fd[0] is the read-end, fd[1] is the write-end

- Windows calls these **anonymous pipes**
- Pipe is a special type of file.
- Inherited by the child
- Must close unused portions of the pipe
UNIX pipe example

```c
#define READ_END 0
#define WRITE_END 1

int fd[2];

create the pipe:
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }

fork a child process:
    pid = fork();

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]);
```

Child inherits the pipe
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("child read %s\n",read_msg);

/* close the write end of the pipe */
close(fd[READ_END]);
Named Pipes

- Named Pipes (termed FIFO) are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems