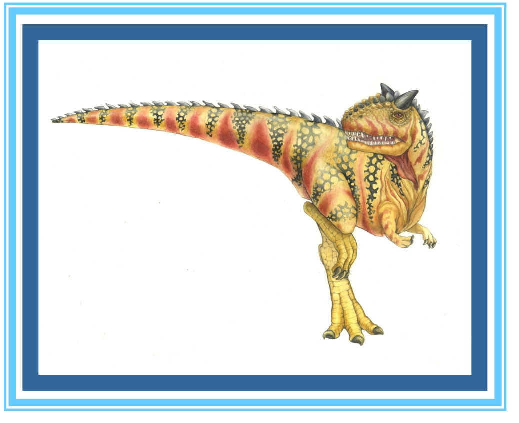
Chapter 3: Processes



Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- 3.7.1 and 3.7.4 are included.

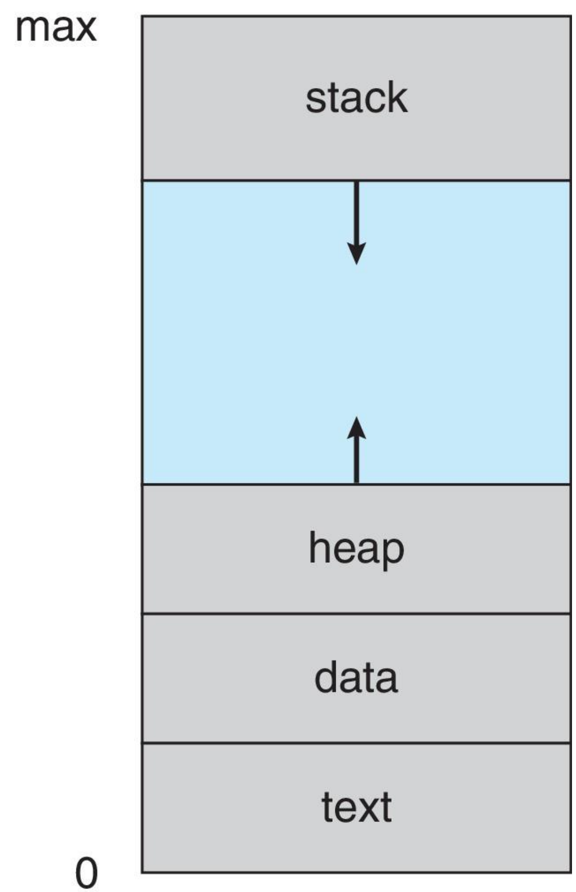
Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

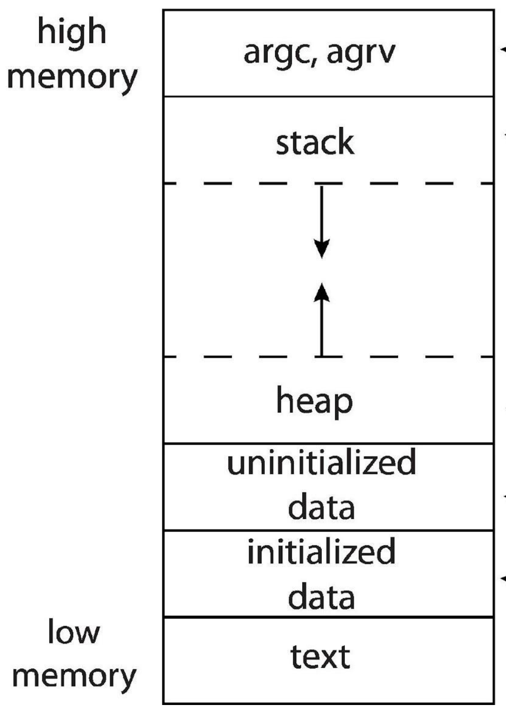
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

Process in Memory



Memory Layout of a C Program

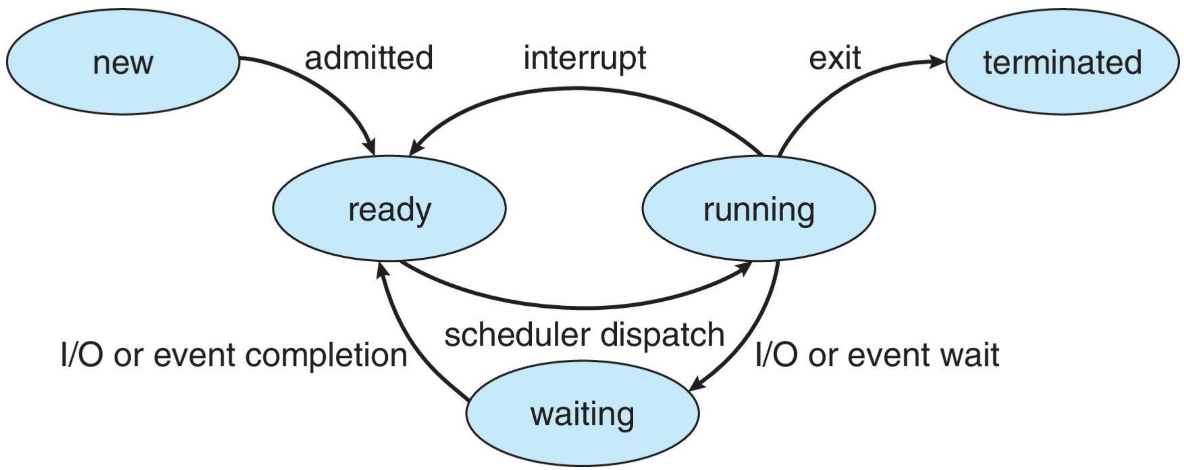



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

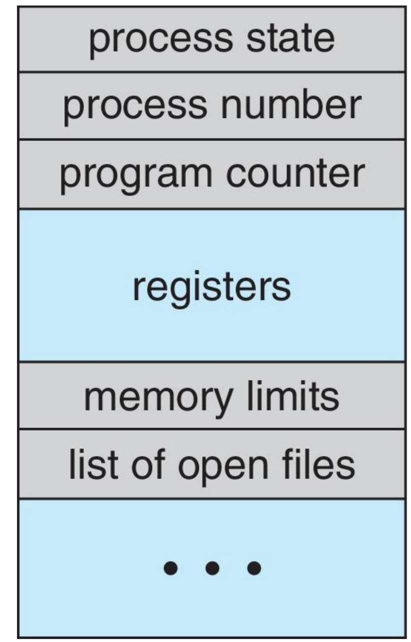
Diagram of Process State



Process Control Block (PCB)

Information associated with each process(also called **task control block**)

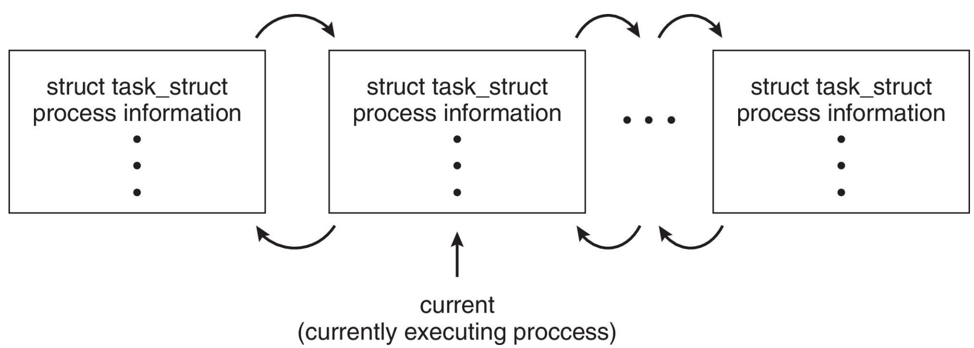
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Process Representation in Linux

Represented by the C structure `task_struct`

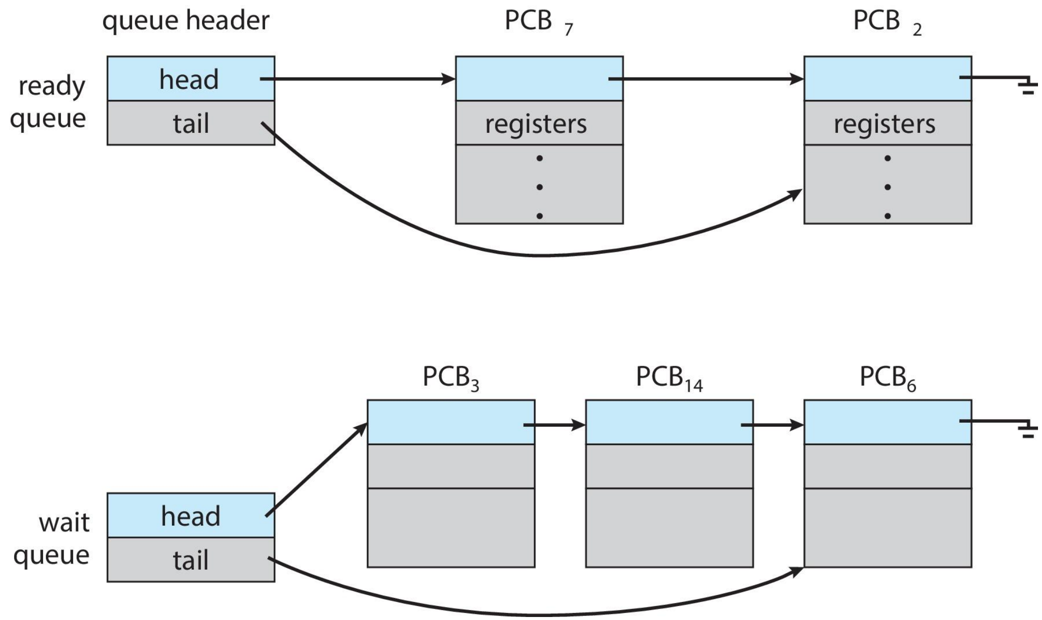
```
pid t_pid;             /* process identifier */
long state;            /* state of the process */
unsigned int time_slice    /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;   /* address space of this process */
```



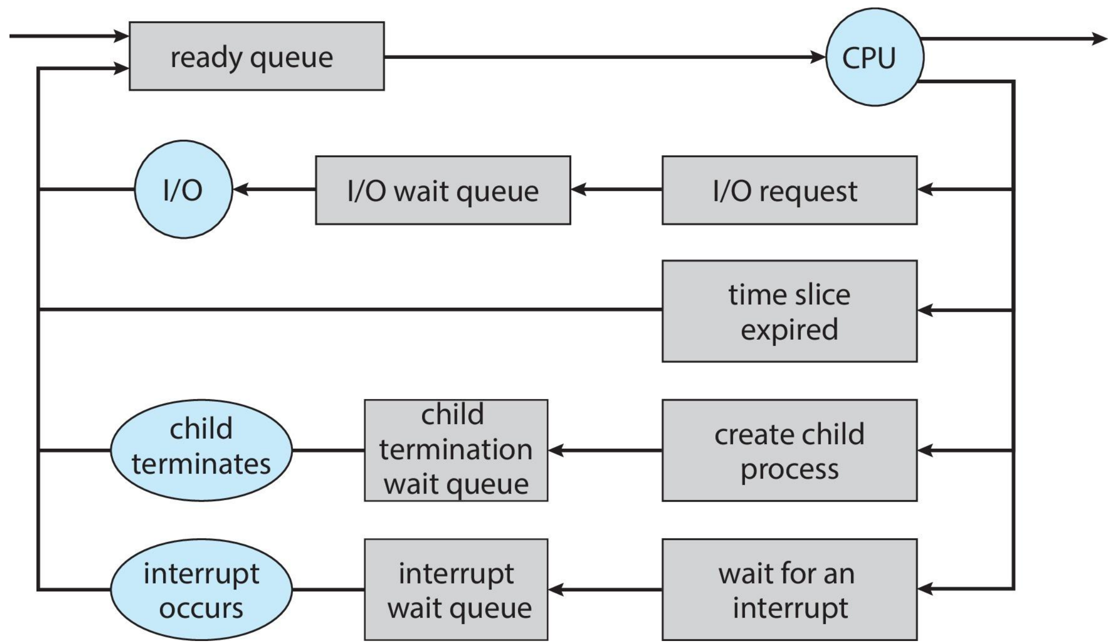
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues

Ready and Wait Queues

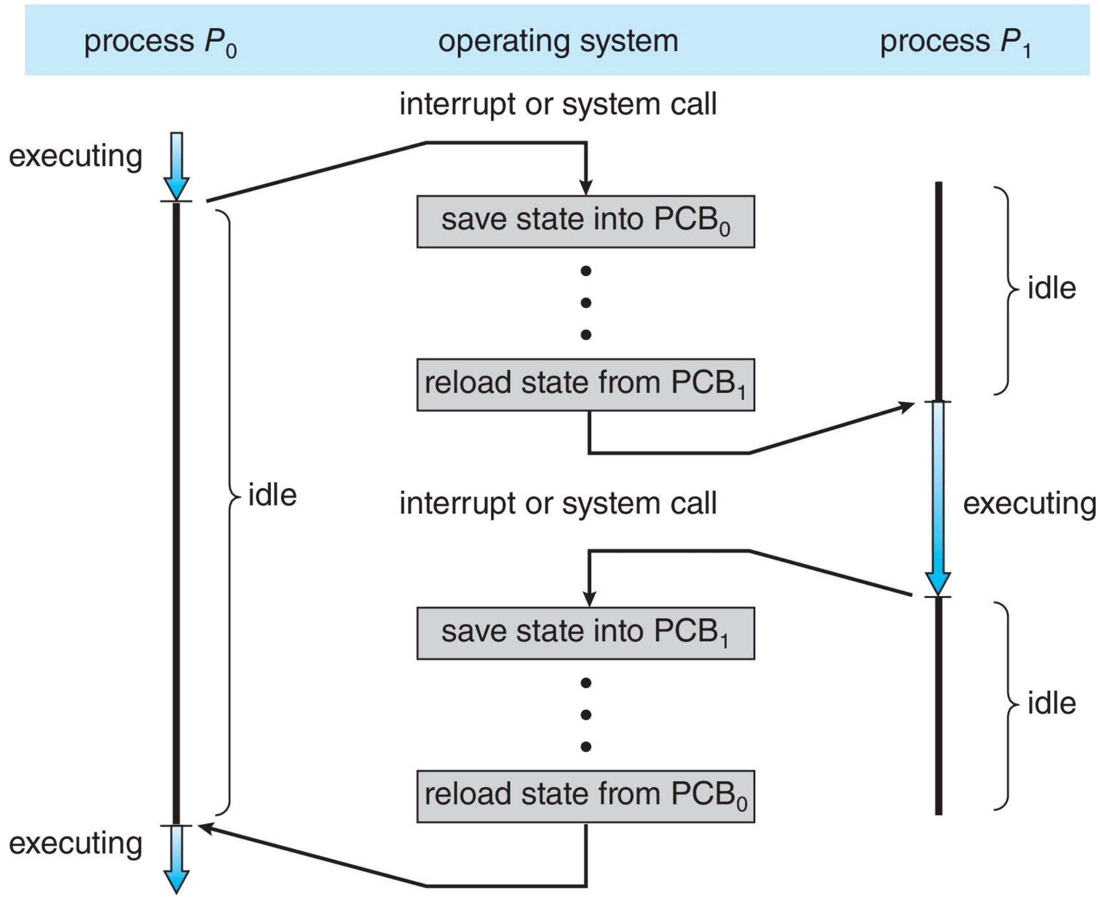


Representation of Process Scheduling



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB □ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU □ multiple contexts loaded at once

Operations on Processes

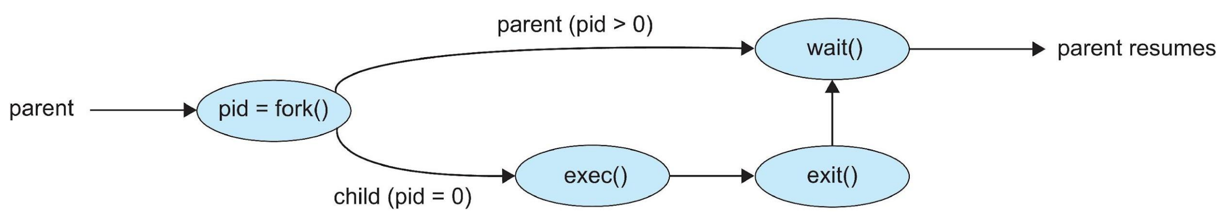
- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

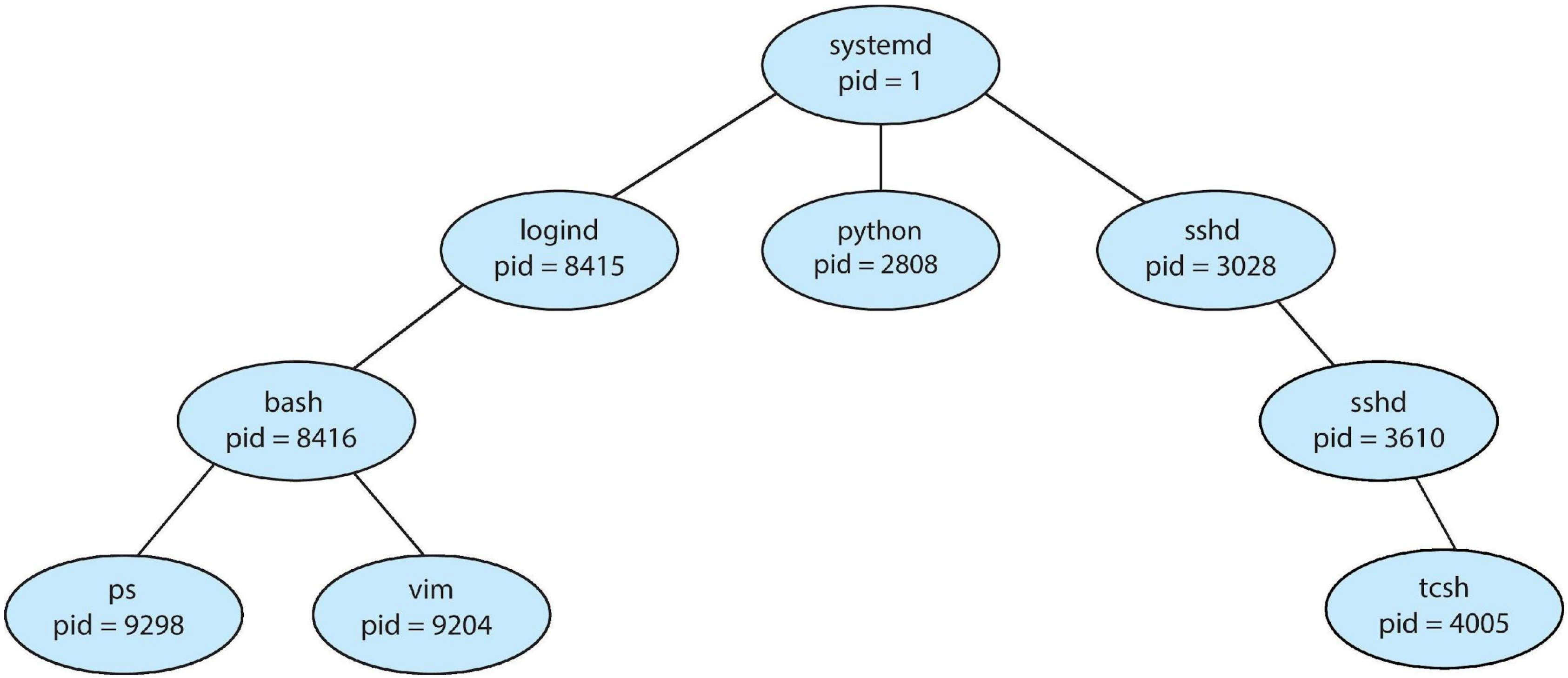
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset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

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
  - Parent process calls **`wait()`** waiting for the child to terminate



A Tree of Processes in Linux



C Program Forking Separate Process

```
#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;
}
```

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 **`abort()`** 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

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

```
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**

Multi Process Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one website 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

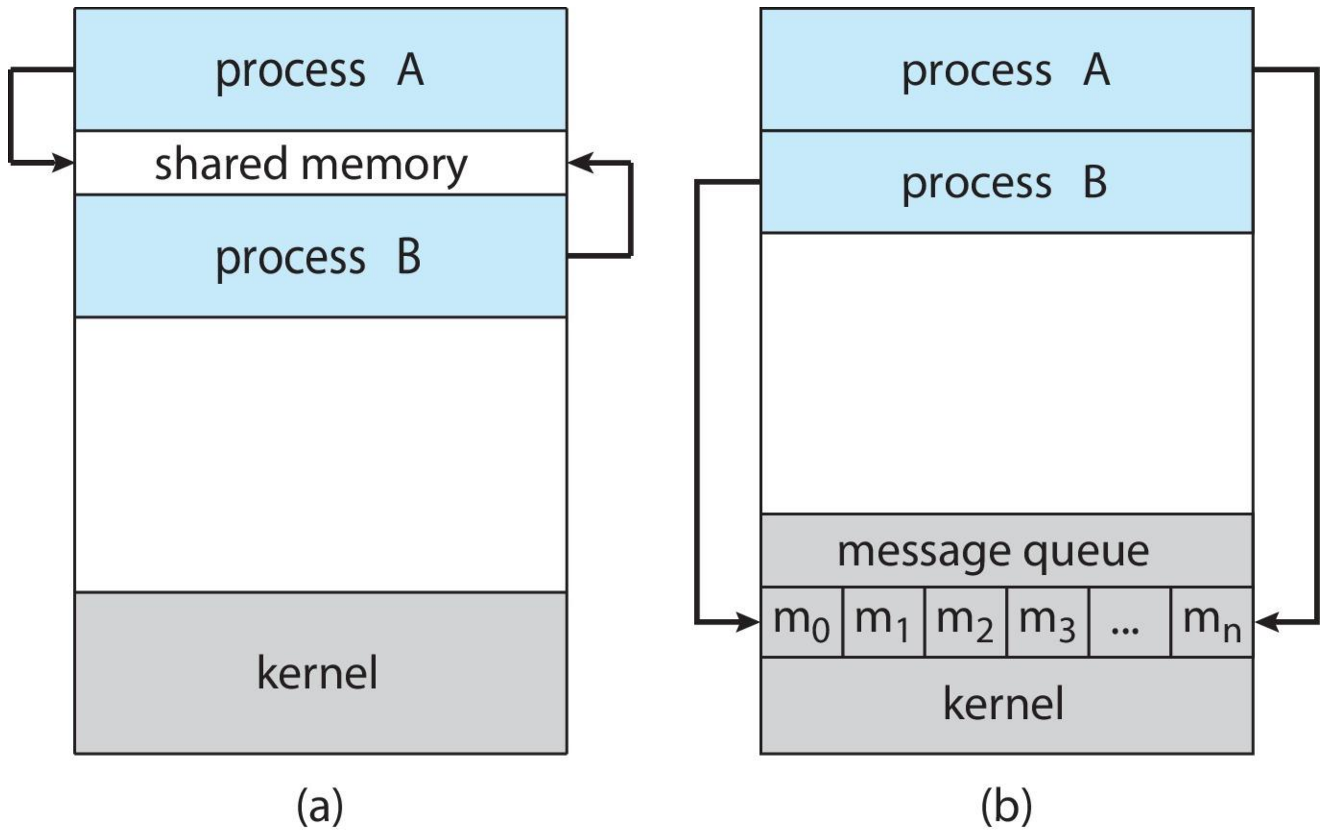
Each tab represents a separate process.

Interprocess Communication

- 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**

Communications Models

(a) Shared memory. (b) Message passing.



Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

Bounded-Buffer – Shared-Memory Solution

Let's look more closely at how the bounded buffer illustrates interprocess communication using shared memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared `buffer` is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when `((in + 1) % BUFFER_SIZE) == out`.

Producer Process – Shared Memory

```
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;
}
```

Figure 3.12 The producer process using shared memory.

Consumer Process – Shared Memory

```
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 */
}
```

Figure 3.13 The consumer process using shared memory.

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object

    `ftruncate(shm_fd, 4096);`
  - Use **`mmap()`** to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by **`mmap()`**.

IPC POSIX Producer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    const int SIZE = 4096;          // the size (in bytes) of shared memory object
    const char *name = "OS";        // name of the shared memory object
    const char *message_0 = "Hello"; // strings written to shared memory
    const char *message_1 = "World!";
    int fd;     // shared memory file descriptor
    char *ptr; // pointer to shared memory obect

    fd = shm_open(name, O_CREAT | O_RDWR, 0666); // create the shared memory object
    ftruncate(fd, SIZE); // configure the size of the shared memory object
    // memory map the shared memory object
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, 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;
}
```

```
gcc test.c -o test -lrt
```

IPC POSIX Consumer

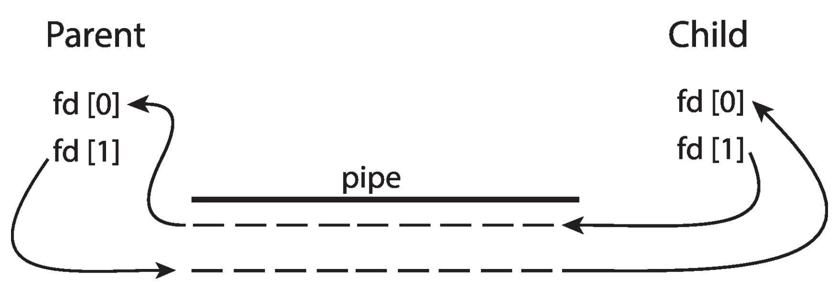
```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    int fd;
    char *ptr;
    fd = shm_open(name, O_RDONLY, 0666);
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    printf("%s", (char *)ptr); // read from the shared memory object
    shm_unlink(name);          // remove the shared memory object
    return 0;
}
```

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – 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**
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**

Ordinary Pipes

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void) {
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    if (pipe(fd) == -1) { // create a pipe
        fprintf(stderr, "Pipe failed");
        return 1;
    }
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    if (pid > 0) { /* parent process */
        close(fd[READ_END]);
        write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
        close(fd[WRITE_END]);
    }
    else { /* child process */
        close(fd[WRITE_END]); // close the unused end of the pipe
        read(fd[READ_END], read_msg, BUFFER_SIZE); // read from the pipe
        printf("read %s", read_msg);
        close(fd[READ_END]); // close the read end of the pipe
    }
    return 0;
}
```

Named Pipes

- Named Pipes 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

Named Pipes (FIFOs)

1. **Persistence**: They are stored as files in the file system and can be accessed by any process as long as they have the necessary permissions. Unrelated processes can communicate using them.
2. **Bidirectional Communication**: Multiple processes can write to and read from the same named pipe. Pipes are unidirectional.
3. **Synchronization**: Named pipes provide a natural synchronization mechanism. Processes can block when reading from an empty pipe or writing to a full pipe, allowing them to synchronize their activities effectively.
4. **Multiple Readers/Writers**: Named pipes allow multiple processes to read from or write to the pipe simultaneously. This feature enables more flexible communication patterns and can be useful in scenarios where multiple processes need to access the same data concurrently.

Named Pipes

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

#define FIFO_FILE "/tmp/myfifo"

int main() {
    int fd;
    char buffer[BUFSIZ];
    ssize_t num_bytes;

    mkfifo(FIFO_FILE, 0666);        // Create the named pipe (FIFO)
    fd = open(FIFO_FILE, O_WRONLY); // Open the named pipe for writing (producer)
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    while (1) { // Producer loop
        printf("Producer: Enter a message (or 'exit' to quit): ");
        fgets(buffer, BUFSIZ, stdin);
        num_bytes = write(fd, buffer, strlen(buffer)); // Write input to the named pipe
        if (num_bytes == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        if (strncmp(buffer, "exit", 4) == 0) { // Check for exit condition
            break;
        }
    }
    close(fd);           // Close the named pipe
    unlink(FIFO_FILE); // Remove the named pipe from the file system

    return 0;
}
```

End of Chapter 3