

Processes

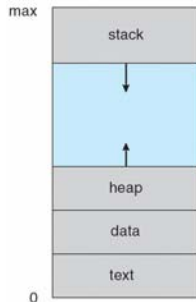
Fan Wu
 Department of Computer Science and Engineering
 Shanghai Jiao Tong University
 Spring 2020

Process Concept

- Process – a program in execution
- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
 - All these activities are *processes*
- Textbook uses the terms *job* and *process* almost interchangeably

The Process

- Multiple parts
 - The program code, also called **text section**
 - **Data section** containing global variables
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Heap** containing memory dynamically allocated during run time

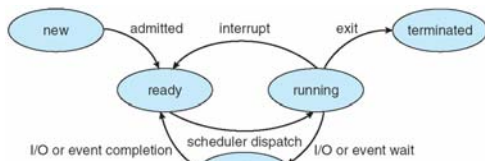


The Process (Cont.)

- What is the difference between program and process?
 - Program is passive entity, process is active
 - Program becomes process when the executable file is loaded into memory
- Execution of program started via GUI clicks, command line entry of its name, etc
- One program can be several processes
 - Consider multiple users executing the same program

Process State

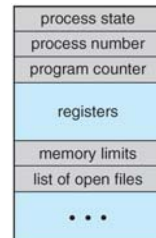
- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution



Process Control Block (PCB)

Information associated with each process

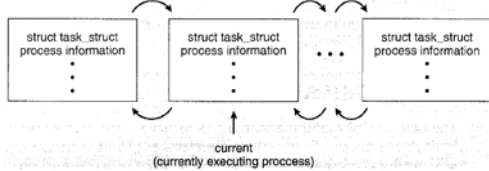
- Process state
- Process number
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



Process Representation in Linux

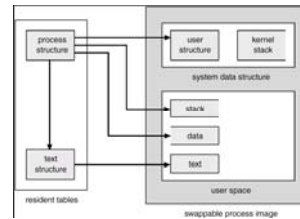
- Represented by the C structure `task_struct`

```
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 pro */
```



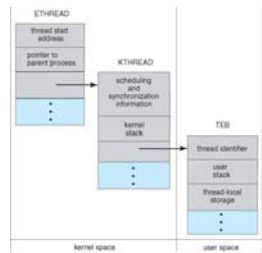
PCBs in UNIX

- The PCB is the box labeled **process structure**, but the **user structure** maintains some of the information as well (only required when the process is resident).

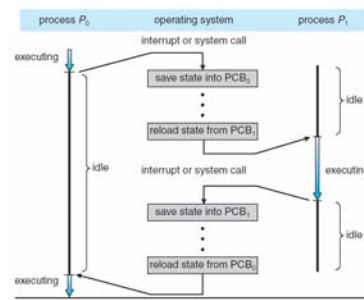


PCBs in Windows NT

- Information is scattered in a variety of objects.
 - Executive Process Block (EPROCESS)
 - Kernel Process Block (KPROCESS)
 - Process Environment Block (PEB)



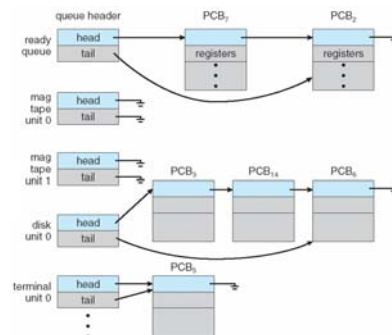
CPU Switch From Process to Process



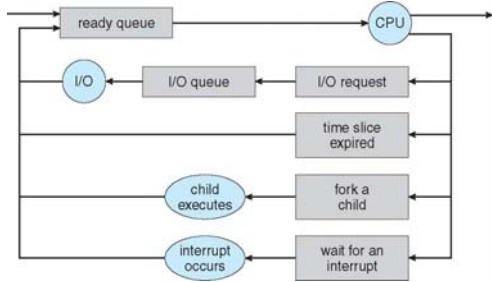
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
 - 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
 - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
 - Sometimes the only scheduler in a system

Schedulers (Cont.)

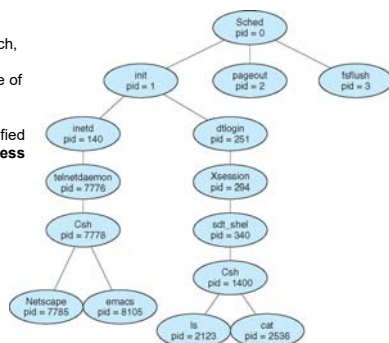
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; some long CPU bursts

Operations on Processes

- 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)**



Process Creation (Cont.)

- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Initialization data
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Address space
 - Child duplicate of parent
 - Child has a program loaded into it

C Program Forking Separate Process

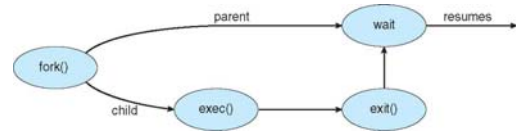
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i = 1;
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        printf("This is child.");
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete.");
    }
    return 0;
}

```

Process Execution



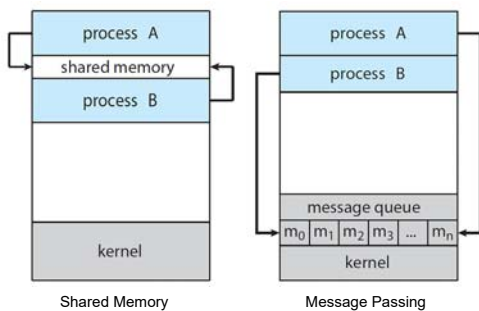
Process Termination

- 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 operating system
- 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 systems do not allow child to continue if its parent terminates
 - All children terminated - **cascading termination**

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

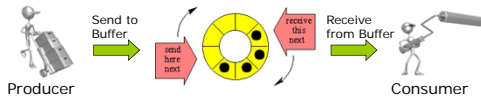


Interprocess Communication – Shared Memory

- A region of memory that is shared by cooperating processes is established.
- Processes can then exchange information by reading and writing data to the shared region.

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


```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;

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

Bounded-Buffer – Shared-Memory Solution

```
while (true) {
    /* Produce an item */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Producer

Consumer

```
while (true) {
    while (in == out)
        ; // do nothing
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

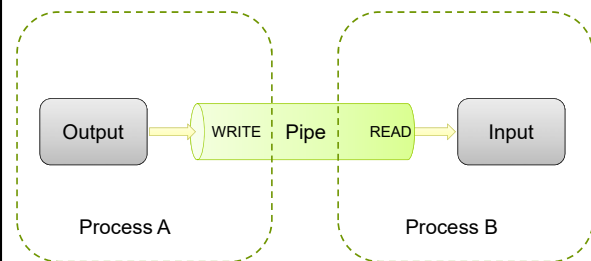
Bounded-Buffer – Shared-Memory Solution

- Weakness:
 - Busy waiting
 - The solution allows only $BUFFER_SIZE - 1$ elements at the same time
- Popquiz:
 - Rewrite the previous processes to allow $BUFFER_SIZE$ items in the buffer at the same time

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 in fact unidirectional
- Require parent-child relationship between communicating processes

Ordinary Pipe



Using Pipe – Part 1

- First, create a pipe and check for errors

```
int mypipe[2];
if (pipe(mypipe)) {
    fprintf(stderr, "Pipe failed.\n");
    return -1;
}
```

mypipe[0]	read-end
mypipe[1]	write-end

- Second, fork your threads
- Third, close the pipes you don't need in that thread
 - reader should close(mypipe[1]);
 - writer should close(mypipe[0]);

Using Pipe – Part 2

- Fourth, the writer should write the data to the pipe
 - write(mypipe[1],&c,1);
- Fifth, the reader reads from the data from the pipe:
 - while (read(mypipe[0],&c,1)>0) {
//do something, loop will exit when WRITER closes pipe
}
- Sixth, when writer is done with the pipe, close it
 - close(mypipe[1]); //EOF is sent to reader
- Seventh, when reader receives EOF from closed pipe, close the pipe and exit your polling loop
 - close(mypipe[0]); //all pipes should be closed now

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) – 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

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; The processes need to know only each other's identity to communicate
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link

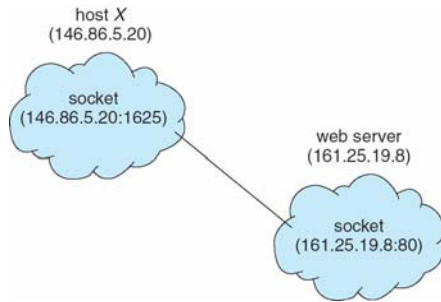
Indirect Communication

- Messages are directed to and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Primitives are defined as:
 - send(A, message) – send a message to mailbox A
 - receive(A, message) – receive a message from mailbox A
- 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

Sockets

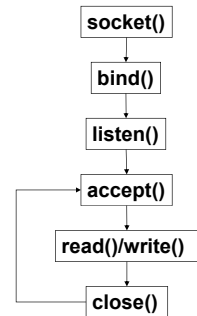
- A socket is defined as an endpoint for communication
- Concatenation of IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets

Socket Communication



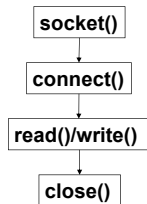
Steps to Create Server Side

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call.
3. Listen for connections with the `listen()` system call
4. Accept a connection with the `accept()` system call (This call typically blocks until a client connects with the server)
5. Send and receive data with `read()` and `write()` system calls
6. Close connection with `close()` system call

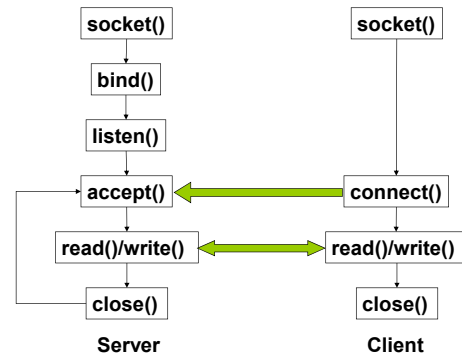


Steps to Create Client Side

1. Create a socket with the `socket()` system call
2. Connect the socket to the address of the server using the `connect()` system call
3. Send and receive data with `read()` and `write()` system calls.
4. Close the socket with `close()` system call



Interaction Between Client and Server



Internet Domain Socket

- IP address:
 - 32 bits (IPv4) or 128 bits (IPv6)
 - C/S work on same host: just use `localhost`
- Port
 - 16 bit unsigned integer
 - Lower numbers are reserved for standard services
- Transport layer protocol: TCP / UDP

Headers

- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <string.h>`
- `#include <sstream>`
- `#include <unistd.h>`
- `#include <sys/types.h>`
 - Definitions of a number of data types used in system calls
- `#include <sys/socket.h>`
 - Definitions of structures needed for sockets
- `#include <netinet/in.h>`
 - Constants and structures needed for internet domain addresses

Creating Socket

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) {
    perror("ERROR opening socket");
    exit(2);
}
```

- AF_INET: address domain
- SOCK_STREAM: stream socket, characters are read in a continuous stream as if from a file or pipe
- 0: protocol. The operating system chooses the most appropriate protocol. It will choose TCP for stream sockets.

Binding Socket

```
struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(BASIC_SERVER_PORT);
bind(sockfd, (sockaddr*)&serv_addr, sizeof(serv_addr));
//error check
```

- INADDR_ANY: get IP address of the host automatically
- htonl, htons: data format conversion
- bind(): binds a socket to an address

Listening and Accepting Connection

```
listen(sockfd, 5);
```

- listen(): allows the server to listen on the socket for connections, with a backlog queue of size 5.

```
int client_sockfd;
struct sockaddr_in client_addr;
int len = sizeof(client_addr);
client_sockfd = accept(sockfd, (sockaddr*)&client_addr,
&len);
//error check
```

- accept(): block process until a client connects to the server. It returns a new socket file descriptor, if the connection is created.

Reading and Writing

```
char buf[1024];
int nread = read(client_sockfd, buf, 1024);
```

read(): reads from the socket

```
write(client_sockfd, buf, len);
```

write(): writes to the socket

```
close(client_sockfd);
```

close(): closes the socket

Connecting A Client to A Server

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
// error check
struct sockaddr_in serv_addr;
struct hostent *host;
serv_addr.sin_family = AF_INET;
host = gethostbyname(argv[1]);
// error check
memcpy(&serv_addr.sin_addr.s_addr, host->h_addr,
host->h_length);
serv_addr.sin_port = htons(BASIC_SERVER_PORT);
connect(sockfd, (sockaddr*)&serv_addr, sizeof(serv_addr))
// error check
```

Homework

- Reading
 - Chapter 3