Processes

Operating Systems

Yu-Ting Wu

Outline

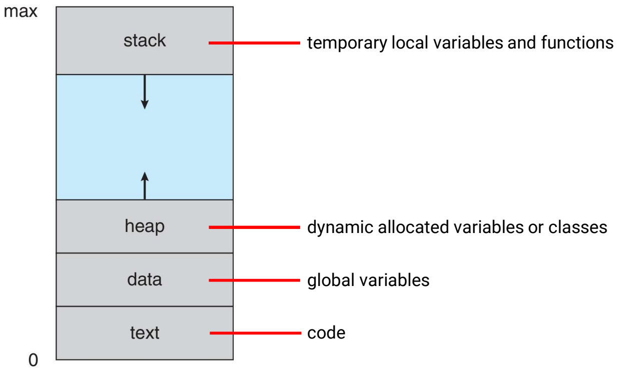
- Process concept
- Process scheduling
- Operations on processes
- Inter-process communication

Process Concept

Process Concept

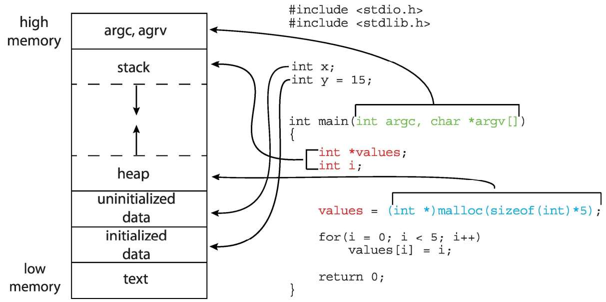
- An operating system concurrently executes a variety of programs
  - **Program: passive entity**, binary file stored **in disk**
  - **Process: active entity**, a running program **in memory**
- A process includes
  - **Code segment** (text section)
  - **Data section:** global variables
  - **Stack:** temporary local variables and functions
  - **Heap:** dynamic allocated variables or classes
  - **Current activity** (e.g., program counter, register contents)
  - **Associated resources** (e.g., handlers of open files)

Process in Memory



Process in Memory (cont.)

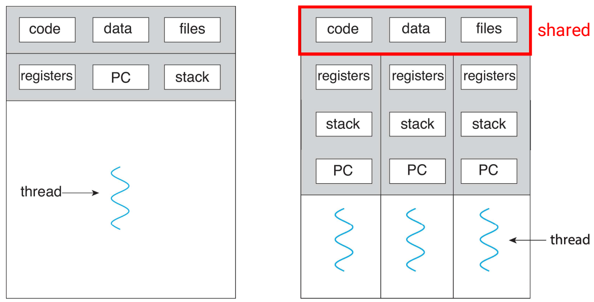
- Example: memory layout of a C program



Thread

- A thread is a **lightweight process**
  - Basic unit of CPU utilization
- All threads belonging to the same process **share**
  - Code section
  - Data section
  - OS resource (open files, signals)
- But each thread has its own
  - Thread ID
  - Program counter
  - Register set
  - Stack

Thread (cont.)

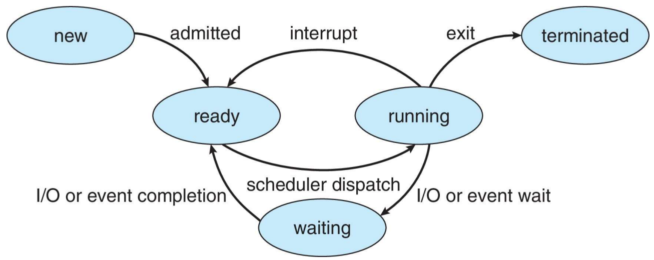


single-threaded process   multithreaded process

Process State

- Types of states
  - **New**
    - The process is being created
  - **Ready**
    - The process is in the memory waiting to be assigned to a processor
  - **Running**
    - The process whose instructions are being executed by CPU
  - **Waiting**
    - The process is waiting for events to occur
  - **Terminated**
    - The process has finished execution

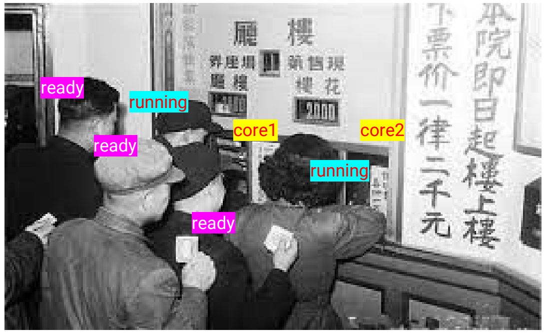
Process State (cont.)



Only one process is running on any processor at any instant

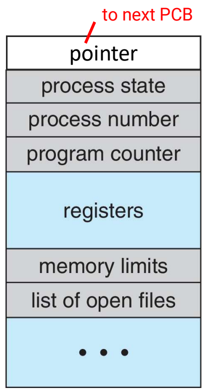
However, many processes may be ready or waiting (put into a queue)

Process State (cont.)



Process Control Block (PCB)

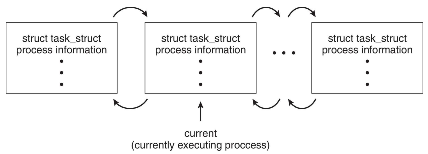
- Store information of each process
  - **Process state**
  - **Program counter**
  - **CPU register**
  - **CPU scheduling information**
    - Priority
  - **Memory management information**
    - base/limit register (loaded into registers while the program is going to the running state)
  - **I/O state information**
  - **Accounting information**



PCB (cont.)

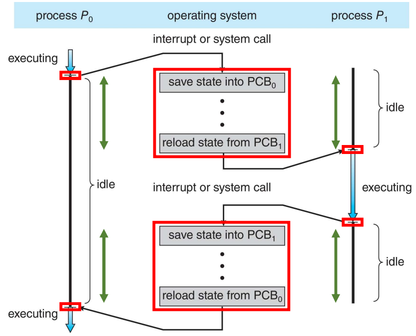
- Process representation in Linux

```
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 */
```



Context Switch

- Occurs when the CPU switches from one process to another



Context Switch (cont.)

- **Context switch**: kernel saves the state of the old process and loads the saved state for the new process
  - The switched context is stored in the PCB
- Context switch time is **purely overhead**
- Switch time (about 1 ~ 1000 ms) depends on
  - Memory speed
  - Number of registers
  - Existence of special instructions
    - Example: a single instruction to save/load all registers
  - Hardware support
    - Example: multiple sets of registers per CPU (multiple contexts loaded at once)

Process Scheduling

Process Scheduling

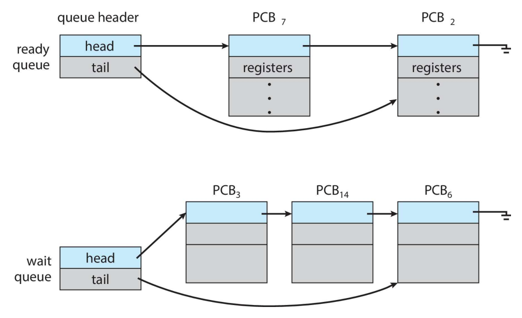
- **Multi-programming**
  - CPU runs process at all times to maximize CPU utilization
- **Time sharing**
  - Switch CPU frequently such that users can interact with each program while it is running
- Process will have to wait until the CPU is free and can be re-scheduled

Process Scheduling Queues

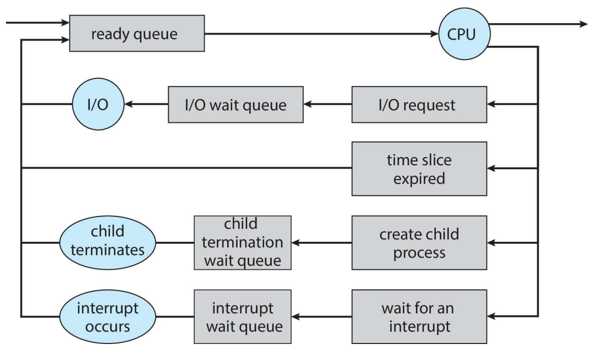
- Maintain **scheduling queues** of processes
  - **Job queue (New state)**
    - Set of all processes in the system
  - **Ready queue (Ready state)**
    - Set of all processes residing in main memory
    - Ready and waiting to execute
  - **Waiting queue (Wait State)**
    - Set of processes waiting for an event (e.g., I/O)
  - Processes migrate among the various queues

Process Scheduling Queues (cont.)

- Ready queue and waiting queue

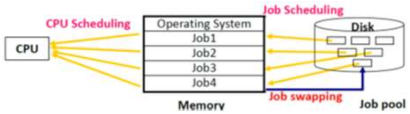


Process Scheduling Queues (cont.)



Process Schedulers

- **Short-term scheduler (CPU scheduler)**
  - Select which process should be executed and allocated CPU **(Ready state ➔ Running state)**
- **Long-term scheduler (job scheduler)**
  - Select which processes should be loaded into memory and brought into the ready queue **(New state ➔ Ready state)**
- **Medium-term scheduler**
  - Select which processes should be swapped in/out memory **(Ready state ➔ Waiting state)**



Long-Term Scheduler

- Control **degree of multi-programming**
- Execute less frequently
  - Invoke only when a process leaves the system or once several minutes
- Strategy
  - Select a **good mix of CPU-bound and I/O bound processes** to increase system overall performance
- New OSes might not contain long-term scheduler
  - The growing memory space
  - Virtual memory (by medium-term scheduler)

Short-Term Scheduler

- Execute quite frequently
  - Example: once per 100 ms.
- Must be efficient
  - If 10 ms for picking a job, 100 ms for such a pick
    ➔ overhead = 10/110 = **9%**
- Must ensure fairness

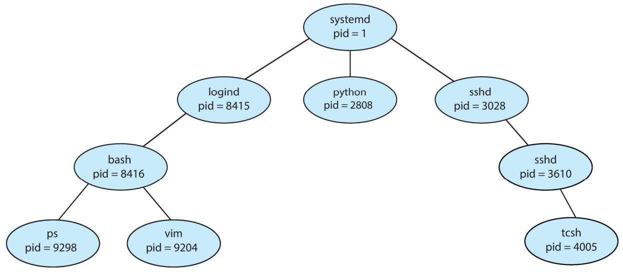
Medium-Term Scheduler

- **Swap out:**
  - Remove processes from memory (to virtual memory) to reduce the degree of multi-programming
- **Swap in:**
  - Reintroduce swap-out processes into memory
- Purpose: improve **process mix** and **free up memory**

Operations on Processes

Tree of Processes

- Each process is identified by a unique **processor identifier (pid)**



Process Creation

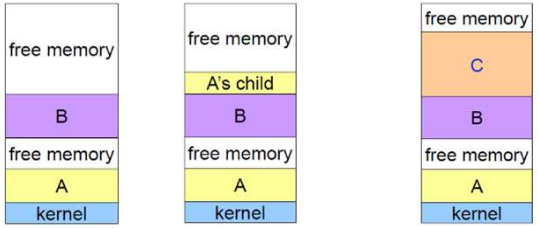
- **Resource sharing** (three possibilities)
  - Parent and child processes share **all** resources
  - Child process shares **subset** of parent's resources
  - Parent and child share **no** resources
- **Execution order** (two possibilities)
  - Parent and children execute **concurrently**
  - Parent **waits until children terminate**
- **Address space** (two possibilities)
  - Children **duplicate** of parent, communication via sharing variables
  - Child **has a program loaded into it**, communication via message passing

UNIX / Linux Process Creation

- **fork system call**
  - Create a new (child) process
  - The new process **duplicates** the address space of its parent
  - Child and parent **execute concurrently** after fork
  - Child: return value of fork is 0
  - Parent: return value of fork is PID of the child process
- **execlp system call**
  - Load a new binary file into memory
  - Destroy the old code
- **wait system call**
  - The parent **waits** for one of its child processes to complete

UNIX / Linux Process Creation (cont.)

- Memory space of fork()
  - Old implementation: A's child is an exact copy of parent
  - Current implementation: use **copy-on-write** technique to store differences in A's child address space



originally — after A does a **fork** — after the child does an **execlp**

UNIX / Linux Example

```
#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;
}
```

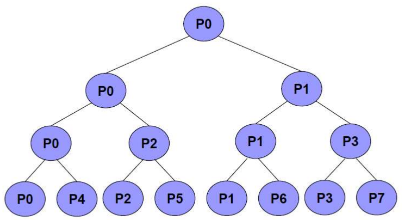
printf("Process End!");

Question:
How many times does "*Process End!*" show?

UNIX / Linux Example Quiz

- How many processors are created?

#include <stdio.h>
#include <unistd..h>
int main()
{
for (int i = 0; i < 3 ; i++)
fork();
return 0;
}



Process Termination

- Terminate when the last statement is executed or **exit()** is called
  - Return status data from child to parent
  - All resources of the process, including physical and virtual memory, open files, I/O buffers, are deallocated by the OS
- Parent may terminate execution of children processes by specifying its PID (**abort**)
  - Children has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the OS does not allow a child to continue if its parent terminates

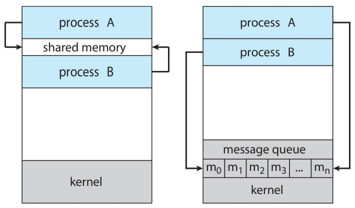
Inter-Process Communication

Inter-Process Communication (IPC)

- **Inter-process communication**
  - A set of methods for the exchange of data among multiple threads in one or more processes
- **Independent process**
  - Cannot affect or be affected by other processes
- **Cooperating process**
  - Otherwise
- Purposes
  - Information sharing
  - Computation speedup
  - Convenience (perform several tasks at one time)
  - Modularity

Communication Methods

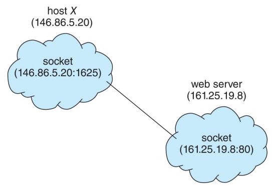
- **Shared memory**
  - Require more careful **user synchronization**
  - Implemented by memory access (faster)
  - Use memory address to access data
- **Message passing**
  - No conflict: more efficient for small data
  - Use send/recv message
  - Implement by system call (slower)



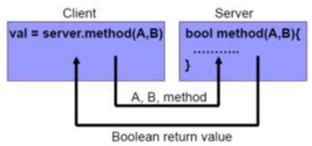
shared memory   message passing

Message Passing Methods

- **Sockets**
  - A network connection identified by **IP** and **port**
  - Exchange **unstructured stream of bytes**



- **Remote Procedure Calls**
  - Cause a procedure to execute in another address space
  - **Parameters** and **return values** are passed by messages

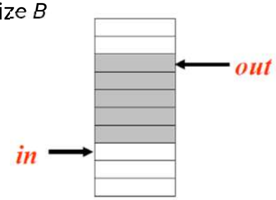


Shared Memory

- Processes are responsible for
  - **Establishing a region of shared memory** (ask OS for help)
    - Typically, the created shared-memory regions resides in the address of the process creating the shared memory segment
    - Participating processes must agree to remove memory access constraint from OS
  - **Determining the form of the data and the location**
  - **Synchronization**: ensuring data are not written simultaneously by processes

Consumer and Producer

- **Producer** process produces information that is consumed by a **Consumer** process
- Buffer as a circular array with size *B*
  - Next free: *in*
  - First available: *out*
  - Empty: *in = out*
  - Full: *(in + 1) % B = out*



- The solution allows at most (B - 1) item in the buffer
  - Otherwise, cannot tell the buffer is empty or full

Consumer and Producer (cont.)

- **Producer** process

```
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;
}
```


Consumer and Producer (cont.)

- **Consumer** process

```
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 */
}
```

Consumer and Producer (cont.)

- Another solution for filling all the buffer
- Use an additional variable, **counter**, for keeping track of the number of items in the buffer
- Initially, counter is set to zero
- Counter is increased by one by the producer after it produces a new item
- Counter is decreased by one by the consumer after it consumes an item

Consumer and Producer (cont.)

- **Producer** process (new version)

```
while (true) {
       /* produce an item in next produced */

       while (counter == BUFFER_SIZE)
              ; /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
       counter++;
}
```


Consumer and Producer (cont.)

- **Consumer** process (new version)

```
while (true) {
       while (counter == 0)
              ; /* do nothing */
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       counter--;
       /* consume the item in next consumed */
}
```


Consumer and Producer (cont.)

- **Race condition**
  - Counter++ can be implemented as
    ```
    register1 = counter
    register1 = register1 + 1
    counter = register1
    ```
  - Counter−− can be implemented as
    ```
    register2 = counter
    register2 = register2 - 1
    counter = register2
    ```
  - Example (initially counter = 5):

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

  - Let's discussed this problem again in Chapter 6

Message Passing System

- Mechanism for processes to **communicate** and **synchronize** their actions
- IPC provides two operations
  - **Send** (message)
  - **Receive** (message)
- To communicate, processes need to
  - Establish a **communication link**
  - Exchange a message via send/receive

Message Passing System (cont.)

- Implementation of communication link
  - **Physical**
    - HW bus
    - Network
  - **Logical (properties of the link)**
    - Direct or indirect communication
    - Symmetric or asymmetric communication
    - Blocking or non-blocking
    - Automatic or explicit buffering
    - Send by copy or send by reference
    - Fixed-sized or variable-sized messages

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**
  - **One-to-one** relationship between links and processes
  - The link may be unidirectional, but is usually **bidirectional**
- **Limited modularity:** if the name of a process is changed, all old names should be found

Indirect Communication

- Messages are directed and received from **mailboxes** (also referred as **ports**)
  - Each mailbox has a unique ID
  - Processes can communicate if they share a mailbox
  - ***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**
  - **Many-to-many** relationship between links and processes
  - Link may be unidirectional or bi-directional
  - Mailbox can be owned either by OS or processes

Indirect Communication (cont.)

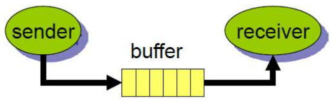
- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1 sends, P2 and P3 receives
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation (by locking and delay)
  - Allow the system to select arbitrary the receiver (sender is notified who the receiver was)

Synchronization

- Messages passing may be either **blocking** or **non-blocking**
- **Blocking (synchronous)**
  - **Blocking send**: sender is blocked until the message is received by receiver or by the mailbox
  - **Blocking receive**: receiver is blocking until the message is available
- **Non-blocking (asynchronous)**
  - **Non-blocking send**: sender sends the message and resumes operation
  - **Non-blocking receive**: receiver receives a valid message or a null

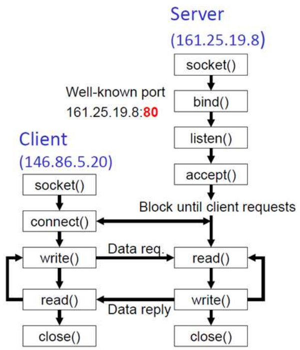
Synchronization (cont.)

- Buffer implementation for queue of messages attached to a link
  - **Zero capacity**
    - No messages are queued on a link
    - Sender must wait for receiver
  - **Bounded capacity**
    - Finite length of *n* messages
    - Sender must wait if the link is full
  - **Unbounded capacity**
    - Infinite length
    - Sender never waits



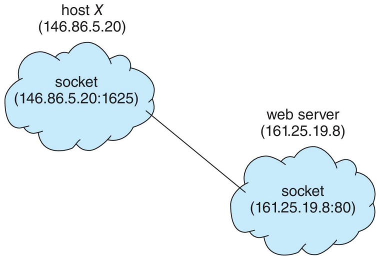
Sockets

- A socket is identified by a concatenation of **IP address** and a **port number**
- Communication consists between a pair of sockets
- Use ***127.0.0.1*** to refer itself



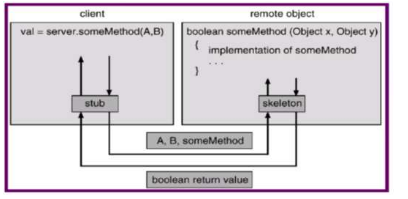
Socket (cont.)

- Consider as a low-level form of communication **unstructured stream of bytes** to be exchanged
- Data parsing responsibility falls upon the server and the client applications



Remote Procedure Calls (RPC)

- Remote procedure call abstracts procedure calls between processes on networked systems
  - Allow programs to call procedures located on other machines (and other processes)
- **Stub/Skeleton**: client-side/server-side proxy for the actual procedure on the server



Remote Procedure Calls (cont.)

- **Client stub**
  - Pack parameters into a message (parameter marshaling)
  - Call OS to send directly to the server
  - Wait for the results returned from the server

- **Server skeleton**
  - Receive a call from a client
  - Unpack the parameters
  - Call the responding procedure
  - Return results to the caller

Objectives Review

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations
- Describe and contrast inter-process communication using shared memory and message passing