How do processes communicate? Is this efficient?

What are possible execution states and how does the system move from one state to another?

How are processes represented in the OS?

A process as the unit of execution.

Today: Process Management

<table>
<thead>
<tr>
<th>System Management, Persistence</th>
<th>File system calls, file sharing</th>
<th>Distributed systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC system calls, distributed file system</td>
<td>Management, Protection, Virtual memory</td>
<td>Memory</td>
</tr>
<tr>
<td>System calls</td>
<td>Management, Protection, Virtual memory</td>
<td>Memory</td>
</tr>
<tr>
<td>Terminal, Mouse, Printer, Memory</td>
<td>Management, Protection, Virtual memory</td>
<td>Memory</td>
</tr>
<tr>
<td>Processor</td>
<td>Management, Protection, Virtual memory</td>
<td>Memory</td>
</tr>
</tbody>
</table>

Example OS Services

From the architecture to the OS to the user: Architectural Resources, OS management, and User Abstractions.

The Big Picture So Far
Example Process State in Memory

- Process execution state (ready, running, etc.)
- A set of OS resources in use (e.g., open files)
- Values of CPU registers
- An execution stack with the program’s call chain (the stack pointer (SP))
- The program counter (PC), indicating the next instruction
- Space for dynamic data (the heap), the heap pointer (HP)
- The static data for the running program
- The code for the running program

Process state consists of at least:

- A process executes sequentially, one instruction at a time
- A process with its own state (e.g., WS Word)
- Several processes may run the same program, but each is a distinct program
- Dynamic execution context of an executing program
The OS manages multiple active processes using state queues.

Example:

As the program executes, it moves from state to state, as a result of the external actions (interrupts), program actions (e.g., system calls), and OS actions (scheduling), and the OS is destroying the process waiting for an event to complete (e.g., I/O).

Execution state of a process indicates what it is doing.

- Waiting
- Ready
- New
- Running
- Terminated
Each I/O device has its own wait queue.

The OS can use different policies to manage each queue.

Its current queue and moved to its new state queue.

When the OS changes the state of a process, the PCB is unlinked from

in the same queue.

The OS places the PCBs of all the processes in the same execution state

The OS maintains the PCBs of all the processes in state queues.

---

The PCB contains:

- The OS deallocates the PCB when the process terminates
- The OS allocates a new PCB on the creation of each process and places it on a state
- The PCB tracks the execution state and location of each process
- The process control block (PCB): OS data structure to keep track of all

---

Process Data Structures

---

Department of Computer Science, Ulster University
The cost of a context switch and the time between switches are closely related.

- Time sharing systems may do 100 to 1000 context switches a second.

- When the OS switches, it saves the current values of the registers.
- While a process is running, the CPU modifies the program counter (PC).
- (PC, SP, etc.) from the PCB
- (PC, SP, etc.) into the PCB
- The OS starts executing a ready process by loading hardware registers.
- Relatively expensive operation.

Starting and stopping processes is called a context switch, and it is a

PCBs and Hardware State

State Queues: Example
First one completes.
with your shell. Otherwise, your next shell command must wait until the

- If you type an & after the command, Unix will run the process in parallel

"exec" (executes) emacs.

For example, you type emacs, the OS "forks" a new process and then

is an implicit fork and exec pair.
Every command you type into the shell is a child of your shell process and
When you log in to a machine running Unix, you create a shell process.

Creating a Process: Example

System call:
- The child then starts a new and different program within itself. Via a call to exec
  continue execution
- The parent can wait for the child to terminate by executing the wait system call or
  * In the child process, the return value is 0
  * In the parent process, fork returns the process id of the child
- The only difference between the child and the parent is the value returned by fork
- Fork copies variables and registers from the parent to the child

In Unix, the fork system call called is used to create child processes

- A parent can either wait for the child to complete, or continue in parallel
- The parent designates (or donates) resources and privileges to its children
- The creator is called the parent and the new process is the child

One process can create other processes to do work.

Creating a Process
gets() reads a line from a file.

waitpid() waits for the named process to finish execution.

sleep() suspends execution for at least the specified time.

execvp() replaces the program of the current process with the named program.

fork() forks a new child process that is a copy of the parent.

---

Example Unix Program: Explanation

```c
{ 

  printf("%s %s
", program, argv[0]); 

  pid_t pid = fork(); 

  if (pid == 0) { 
    printf("I am the child process\n"); 
  } else { 
    printf("I am the parent process\n"); 
  }

  waitpid(pid, 0, 0); 

  if (pid == 0) { 
    printf("I am the child process\n"); 
  } else { 
    printf("I am the parent process\n"); 
  }

} 
```

---

Example Unix Program: Fork
- A process can terminate a child using the `kill` system call.
- A process can terminate itself using the `exit` system call.

  In Unix process
  
  On process termination, the OS reclaim all resources assigned to the

---

**What is happening on the Fork**

---

**Process Termination**
Processes

In these machines, we must cooperate and coordinate between separate
processes, which each is smaller than a single monolithic program, and
— easily share information between tasks.
— cooperate in application to achieve a better program structure as a set of cooperating
— improve performance by overlapping activities or performing work in parallel.

Cooperating processes can

Cooperating processes work with each other to accomplish a single task.

Any two processes are either independent or cooperating.
... send (nextp, consumer)...
consume item nextc
... produce item nextp
repeat
receive (nextc, producer)
... repeat
producer

else consumer():
if (work() \( i = 0 \) producer()):
... main()

Communication using Message Passing

Producers and consumers can communicate using message passing or shared memory

... { ... 
consume nextc
... 
out = out+1 \mod n
nextc = buffer[out]
while in = out \mod n do
out = out+1 \mod n
while in = out \mod n do
nextp = produced item
repeat forever
consumer
... }

Producer
out = 0
in = 0
max outstanding items
... 
repeat forever
consumer
... 

Communication using Shared Memory

multiplex consumers?
How would you use message passing to implement a single producer and
OS keeps track of messages (copies them, notifies receiving process, etc.).

• consumer, out from consumer to producer.
• communication of the in and out variables (in from producer to
• A bounded buffer would require the tests in the previous slide, and

• the consumer is assumed to have an infinite buffer size.
• each process needs to be able to name the other process.
• distributed systems typically communicate using message passing

Message Passing
Processes communicate either with message passing or shared memory

context switch

The program currently executing on the CPU is changed by performing a

0: a uniprocessor, there is at most one running process at a time.

A process is either New, Ready, Waiting, Running, or Terminated.

- PCBs contain process state, scheduling and memory management information, etc
- Processes are represented as Process Control Blocks in the OS

A process is the unit of execution.

Process Management: Summary

... in = in+1 mod n
consumer item next = buffer[in]
... while in+1 mod n = out do nop
next = buffer[in]
... while in = out do nop
repeat
consumer
producer
end

if consumer() end
if (fork()) producer() end
out = 0
in = 0
map(..., our PROT-WRITE, MAP-SHARED, ...)
main()