Processes

CSE 2431: Introduction to Operating Systems
Reading: Chap. 3, [OSC]
Outline

• What Is A Process?
• Process States & PCB
• Process Memory Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
  - Many users may launch same program
  - One user may launch many instances of the same program
- Processes: an executing program

*Question: Real life analogy?*
Real Life Analogy?

• One example:
  – Recipes $\Leftrightarrow$ Programs
    • Step 1: xxx
    • Step 2: xxx
    • Step 3: xxx
    • Step 4: xxx
  – Making dishes $\Leftrightarrow$ Processes
    • Action

• Others? (Group discussion)
Windows Task Manager

Unix Processes: `ps`

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>00:00:04</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>00:00:00</td>
<td>kthreadd</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/0</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
<td>00:00:02</td>
<td>ksoftirqd/0</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/0</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>00:00:00</td>
<td>watchdog/0</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/1</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/1</td>
</tr>
<tr>
<td>9</td>
<td>?</td>
<td>00:00:02</td>
<td>ksoftirqd/1</td>
</tr>
<tr>
<td>10</td>
<td>?</td>
<td>00:00:00</td>
<td>watchdog/1</td>
</tr>
<tr>
<td>11</td>
<td>?</td>
<td>00:00:03</td>
<td>migration/2</td>
</tr>
<tr>
<td>12</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/2</td>
</tr>
<tr>
<td>13</td>
<td>?</td>
<td>00:00:04</td>
<td>ksoftirqd/2</td>
</tr>
<tr>
<td>14</td>
<td>?</td>
<td>00:00:00</td>
<td>watchdog/2</td>
</tr>
<tr>
<td>15</td>
<td>?</td>
<td>00:00:03</td>
<td>migration/3</td>
</tr>
<tr>
<td>16</td>
<td>?</td>
<td>00:00:00</td>
<td>migration/3</td>
</tr>
<tr>
<td>17</td>
<td>?</td>
<td>00:00:03</td>
<td>ksoftirqd/3</td>
</tr>
<tr>
<td>18</td>
<td>?</td>
<td>00:00:00</td>
<td>watchdog/3</td>
</tr>
<tr>
<td>19</td>
<td>?</td>
<td>00:00:17</td>
<td>events/0</td>
</tr>
<tr>
<td>20</td>
<td>?</td>
<td>00:00:15</td>
<td>events/1</td>
</tr>
<tr>
<td>21</td>
<td>?</td>
<td>00:00:12</td>
<td>events/2</td>
</tr>
<tr>
<td>22</td>
<td>?</td>
<td>00:00:18</td>
<td>events/3</td>
</tr>
</tbody>
</table>
So What Is A Process?

- *It’s one executing instance of a “program”*
- It’s separated from other instances
- It can start (“launch”) other processes
- It can be launched by other processes
Outline

• What Is A Process?
• **Process States & PCB**
• Process Memory Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
Process Control Block (PCB): Why?

- PCB contains information associated with a given process:
  - Process state
  - Process identification (Pid)
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
  - Pid of parent process
Process Descriptor in Linux (I)

- struct task_struct
- unsigned long state;
- int prio;
- unsigned long policy;
- struct task_struct *parent;
- struct list_head tasks;
- pid_t pid;
- ...

process descriptor

the task list
Process Descriptor in Linux (II)

- In `<linux/sched.h>`:
  https://github.com/torvalds/linux/blob/master/kernel/sched/sched.h
- In `<asm/thread_info.h>`:
  https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/thread_info.h
As a process executes, it changes its *state*:
- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting** (*blocked*): The process is waiting for some event to occur.
- **Ready**: The process is waiting to be assigned to a CPU.
- **Terminated**: The process has finished execution.
Questions

• Using “working on a lab assignment” as an example
  – What corresponds to “running”?
  – What corresponds to “waiting”?
  – What corresponds to “ready”??
So What Is A Process?

• It’s one executing instance of a “program”
• *It’s separated from other instances*
• It can start (“launch”) other processes
• It can be launched by other processes
What Does This Program Do?

```c
int myval;
int main(int argc, char *argv[])
{
    myval = atoi(argv[1]);
    while (1)
    {
        printf("myval is %d, loc 0x%lx\n", myval, (long) &myval);
    }
}
```

- Now simultaneously start two instances of this program
  - myval 5
  - myval 6

- **What will the outputs be?**
Here’s The Output
Instances Of Programs

• The address was always the same
• The values were different
  – Implies that the processes aren’t seeing each other
  – But they think they’re using the same address

• Conclusion: addresses are not absolute/physical
• Implication: memory mapping
• What’s the benefit?
Outline

• What Is A Process?
• Process States & PCB
• **Process Memory Layout**
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Remember This?

- Application
  - Libraries
  - Portable OS Layer
    - Machine-dependent layer

- User space/level
- Kernel space/level
Address Space (1)

- One (common) approach
  - Kernel is high memory
  - User is low memory

- What restrictions apply?

- `read(f, buf, nbytes)`
Address Space (2)

• Program segments
  – Text
  – Data
  – Stack
  – Heap

• What is text?
• What is data?
• What is stack?
• What is heap?
One Common Layout

- Lots of flexibility
  - Allows stack growth
  - Allows heap growth
  - No predetermined division
Outline

• What Is A Process?
• Process States & PCB
• Process Memory Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Remember This?

- new
- admitted
- interrupt
- exit
- terminated
- ready
- running
- waiting
- I/O or event completion
- scheduler dispatch
- I/O or event wait
Process Scheduling Queues

- **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**
Process Scheduling

- ready queue
- I/O
- I/O queue
- I/O request
- time slice expired
- child executes
- fork a child
- interrupt occurs
- wait for an interrupt
Schedulers

• **Long-term scheduler** (or *job scheduler*)
  – Selects which processes should be brought into the ready queue
  – Invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  – Controls the *degree of multiprogramming*
  – Should balance different types of processes:
    • *I/O-bound process*: spends more time doing I/O than computations, many short CPU bursts
    • *CPU-bound process*: – spends more time doing computations; few very long CPU bursts
• **Short-term scheduler** (or *CPU scheduler*)
  – Selects which process should be executed next and allocates CPU
  – Invoked very frequently (milliseconds) ⇒ (must be fast)
Addition of Medium Term Scheduling

Diagram:
- Swap in
- Partially executed swapped-out processes
- Ready queue
- I/O
- I/O waiting queues
- CPU
- Swap out
- End
Outline

• What Is A Process?
• Process States & PCB
• Process Memory Layout
• Process Scheduling
• **Context Switch**
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Context Switch

- Switch CPU from one process to another
- Performed by scheduler (**dispatcher**)
- It includes:
  - Save PCB state of the old process;
  - Load PCB state of the new process;
  - Flush memory cache;
  - Change memory mapping (TLB);
- **Context switch is expensive** (1–1000 µs)
  - No useful work is done (pure overhead)
  - Can become a bottleneck
  - *Real life analogy?*
CPU Switch From Process to Process

Diagram:
- **Process $P_0$**
  - Executing
  - Interrupt or system call
  - Save state into PCB$_0$
  - ... (omitted)
  - Reload state from PCB$_1$
  - Idle
- **Operating System**
  - Interrupt or system call
  - Save state into PCB$_1$
  - ... (omitted)
  - Reload state from PCB$_0$
- **Process $P_1$**
  - Executing
Outline

• What Is A Process?
• Process States & PCB
• Process Memory Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Process Creation

• System initialization
  – Reboot

• Process creation
  – System call: fork()

• Users request to create a new process
  – Command line or click an icon

• Initiation of a batch job
  – Cron
  – Try “crontab” or “at”
Process Termination

• Normal exit (voluntary)
  – End of main()

• Error exit (voluntary)
  – exit(2)

• Fatal error (involuntary)
  – Divide by 0, core dump

• Killed by another process (involuntary)
  – kill procID, end task
Process Hierarchies

- Parent creates a child process, a child process can create its own processes
Unix Process Related System Calls

int fork(void);

– A new process is created that is an exact copy of the process making the system call, i.e., a copy of the calling process is made and it runs as a new process. The copy is of the memory image of the calling process at the moment of the fork system call, not the program the calling process was started from.

– The new process does not start at the beginning but at the exact point of the fork system call.

– Thus, right after the fork there are two processes (the issuing process and newly created process) with identical memory images.

– The return value to the parent is the process identifier of the new process. The return value to the child process is 0.
Processes after `fork()`
Unix Process Related System Calls

```c
int execv(char *programName, char *argv[]);
```

The program `programName` is loaded in the calling process address space. Thus, the old program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector `argv`, which is an array of strings, that is, an array of pointers to characters. (There are 6 versions of `exec` system call)

```c
void exit(int returnCode);
```

This system call causes a process to exit. The `returnCode` is returned to the parent process if it waits for its child process to terminate.

```c
int wait(int *returnCode);
```

This system call will cause the calling process to wait until *any* process created by the calling process exits. The return value is the process identifier of the process that exited. The return code is stored in `returnCode`. 
Unix Processes: Example 1

What’s the functionality of the following code snippet?

```c
ProgA
{
    int x; char *arg[1]={0};
    x = fork();
    if (x==0) execv("ProgB",arg);
    exit(3)
}
```
Unix Processes: Example 2

Write C code for a process $A$ which creates another process with code from $ProgB$ file, then the process $A$ waits for its child process to terminate, before it exits.

```c
ProgA
{
    int x,y,z; char *arg[1]={0};
    x=fork();
    if (x==0) execv("ProgB",arg);
    y=wait(&z);
    exit(3)
}

ProgB
{
    // -
    exit(5); /*point A*/
    // -
    exit(1); /*point B*/
}
```

If the child process exits at point A, then $z=5$, while if it exits at point B then $z=1$. 
void main (int argc, char *argv [])
{
    int pid;
    pid = fork (); /* fork another process*/

    if (pid <0) { /* error*/
        fprintf (stderr, “Fork failed”);
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execvp (“/bin/ls”, “ls”, NULL);
    }
    else { /* parent process */
        wait(NULL); /* waits for child to terminate
        printf (“Child completed”);
        exit(0);
    }
}

Example from Figure 3.10 in the textbook; it illustrates how Unix commands (e.g., ls command) can be issued from a program.
Unix Processes: Example 4

Write C code for a process A which creates another process to execute code from file “gcc (a compiler)” with a list of 3 parameters passed. Then the process A waits for its child process to terminate, before it exits.

```c
ProgA
{
    int x,y,z;
    char *argv[4]="gcc","-c","NameFileToCompile", 0;

    x = fork();
    if (x==0) execv("gcc",argv);
    y = wait(&z);
    exit(0);
}
```
Outline

- What Is A Process?
- Process States & PCB
- Process Memory Layout
- Process Scheduling
- Context Switch
- Process Operations
- Inter-Process Communication
- Client-Server Communication
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

• Why cooperating?
  – Information sharing
  – Computation speed-up
  – Modularity
  – Convenience
Example: Producer-Consumer Problem

- Paradigm for cooperating processes: *producer* process produces information that is consumed by a *consumer* process
Inter-Process Communication (IPC)

- Mechanisms for processes to communicate and to synchronize their actions
- Shared Memory Systems
  - A region of memory are shared among processes
- Message-Passing Systems
  - Message exchange for communication
- Comparison between shared memory and message-passing
  - Shared memory is more efficient
  - Message-passing is easy for programming
  - More?
Communication Models

(a) and (b) illustrate different communication models between processes A and B. The models depict the interaction between the processes, with arrows indicating data flow. The kernel layer is shown at the bottom, with the user processes (A and B) above it.
Message-Passing Systems

• 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
• Implementation of communication link
  – Physical (e.g., shared memory, hardware bus)
  – Logical (e.g., logical properties)
Implementation Questions

• How are links established?
• Can a link be associated with more than two processes?
• How many links can be there 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?
Direct Communication

• Processes must name each other explicitly:
  – $\text{send}(P, \text{message})$ – send a message to process $P$
  – $\text{receive}(Q, \text{message})$ – 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 (1)

- Messages 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 (2)

• Operations
  – Create a new mailbox
  – Send and receive messages through mailbox
  – Destroy a mailbox

• Primitives are defined as:
  send\((A, \text{message})\) – send a message to mailbox \(A\)
  receive\((A, \text{message})\) – receive a message from mailbox \(A\)
Indirect Communication (3)

- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A
  - $P_1$ sends a message; $P_2$ and $P_3$ receive
  - 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
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Synchronization

• Message passing may be either blocking or non-blocking
• **Blocking is considered synchronous**
  – *Blocking send* has the sender block until the message is received
  – *Blocking receive* has the receiver block until a message is available
• **Non-blocking is considered asynchronous**
  – *Non-blocking send* has the sender send the message and continue
  – *Non-blocking receive* has the receiver receive a valid message or null
Buffering

• Queue of messages attached to the link; implemented in one of the three ways
  – Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  – Bounded capacity – finite length of $n$ messages
    Sender must wait if link full
  – Unbounded capacity – infinite length
    Sender never waits
Outline

• What Is A Process?
• Process States & PCB
• Process Memory Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication
Client-Server Communication

- Why are IPC mechanisms not enough?
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)
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

host X
(146.86.5.20)

socket
(146.86.5.20:1625)

web server
(161.25.19.8)

socket
(161.25.19.8:80)
Remote Procedure Calls

• **Why?**

• Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

• **Stubs**: client-side proxy for the actual procedure on the server.

• The client-side stub locates the server and *marshals* the parameters.

• The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
Execution of RPC

- User calls kernel to send RPC message to procedure X.
- Kernel sends message to matchmaker to find port number.
- Kernel places port P in user RPC message.
- Kernel sends RPC.
- Kernel receives reply, passes it to user.
- From: client To: server Port: matchmaker Re: address for RPC X
- Matchmaker receives message, looks up answer.
- Matchmaker replies to client with port P.
- Daemon listening to port P receives message.
- Daemon processes request and processes send output.
- From: client To: server Port: P <contents>
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPC.
- RMI allows a Java program on one machine to invoke a method on a remote object.

![Remote Method Invocation Diagram](image-url)
Marshaling Parameters

val = server.someMethod(A, B)

boolean someMethod (Object x, Object y)
{
    implementation of someMethod
    ...
}

A, B, someMethod

boolean return value
Summary

• Process Concepts
• Process States & PCB
• Address Space Layout
• Process Scheduling
• Context Switch
• Process Operations
• Inter-Process Communication
• Client-Server Communication