Processes

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Today’s Topics

- What is the process?
- How to implement processes?
- Inter-Process Communication (IPC)
Process Concept (1)

What is the process?

• An instance of a program in execution.
• An encapsulation of the flow of control in a program.
• A dynamic and active entity.
• The basic unit of execution and scheduling.
• A process is named using its process ID (PID).
• Job, task, or sequential process
• A process includes:
  – CPU contexts (registers)
  – OS resources (memory, open files, etc.)
  – Other information (PID, state, owner, etc.)
Process Concept (2)

- Process in memory

<table>
<thead>
<tr>
<th>Segment Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-only segment</td>
<td>(.init, .text, .rodata)</td>
</tr>
<tr>
<td>read/write segment</td>
<td>(.data, .bss)</td>
</tr>
<tr>
<td>run-time heap</td>
<td>(managed by malloc)</td>
</tr>
<tr>
<td>user stack</td>
<td>(created at runtime)</td>
</tr>
<tr>
<td>kernel virtual memory</td>
<td>(code, data, heap, stack)</td>
</tr>
</tbody>
</table>

0xffffffff

0

brk

stack pointer

0xffffffffff
Process Creation (1)

- **Process hierarchy**
  
  - One process can create another process: *parent-child relationship*
  
  - UNIX calls the hierarchy a “process group”
  
  - Windows has no concept of process hierarchy.

  - Browsing a list of processes:
    - `ps` in UNIX
    - `taskmgr` (Task Manager) in Windows

```bash
$ cat file1 | wc
```

```
sh
  cat
  wc
```
Process Creation (2)

- Process creation events
  - Calling a system call
    - `fork()` in POSIX, `CreateProcess()` in Win32
    - Shells or GUIs use this system call internally.
  - System initialization
    - `init` process

- Background processes
  - Do not interact with users
  - Daemons
Process Creation (3)

- **Resource sharing**
  - Parent may inherit all or a part of resources and privileges for its children
    - UNIX: User ID, open files, etc.

- **Execution**
  - Parent may either wait for it to finish, or it may continue in parallel.

- **Address space**
  - Child duplicates the parent’s address space or has a program loaded into it.
Process Termination

- Process termination events
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
    - Exceed allocated resources
    - Segmentation fault
    - Protection fault, etc.
  - Killed by another process (involuntary)
    - By receiving a signal
The `fork()` function is defined in the `<sys/types.h>` header file and the `<unistd.h>` header file. It is used to create a new process from an existing process. The function returns a zero value if it creates a new process, and a positive value if it does not.

The code snippet below demonstrates the use of `fork()` in a C program:

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int pid;

    if ((pid = fork()) == 0) /* child */
        printf("Child of %d is %d\n", getppid(), getpid());
    else /* parent */
        printf("I am %d. My child is %d\n", getpid(), pid);
}
```
fork(): Example Output

% . ./a.out
I am 31098. My child is 31099.
Child of 31098 is 31099.

% . ./a.out
Child of 31100 is 31101.
I am 31100. My child is 31101.
int main()
{
    while (1) {
        char *cmd = read_command();
        int pid;
        if ((pid = fork()) == 0) {
            /* Manipulate stdin/stdout/stderr for pipes and redirections, etc. */
            exec(cmd);
            panic(“exec failed!”);
        } else {
            wait (pid);
        }
    }
}
Process State Transition (1)
Process State Transition (2)

- Linux example

- R: Runnable
- S: Sleeping
- T: Traced or Stopped
- D: Uninterruptible Sleep
- Z: Zombie
- <: High-priority task
- N: Low-priority task
- S: Session leader
- +: In the foreground process group
- I: Multi-threaded
Process Data Structures

- PCB (Process Control Block)
  - Each PCB represents a process.
  - Contains all of the information about a process
    - Process state
    - Program counter
    - CPU registers
    - CPU scheduling information
    - Memory management information
    - Accounting information
    - I/O status information, etc.
  - task_struct in Linux
    - 1456 bytes as of Linux 2.4.18
### Process Control Block (PCB)

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PCBs and Hardware State

- **When a process is running:**
  - Its hardware state is inside the CPU:
    - PC, SP, registers

- **When the OS stops running a process:**
  - It saves the registers’ values in the PCB.

- **When the OS puts the process in the running state:**
  - It loads the hardware registers from the values in that process’ PCB.
Context Switch (1)

**Diagram:**

- **Process $P_0$:**
  - Executing
  - Interrupt or system call
    - Save state into PCB
    - Reload state from PCB

- **Operating System:**
  - Idle

- **Process $P_1$:**
  - Executing
  - Interrupt or system call
    - Save state into PCB
    - Reload state from PCB
  - Idle
Context Switch (2)

- **Context switch (or process switch)**
  - The act of switching the CPU from one process to another.
  - Administrative overhead
    - saving and loading registers and memory maps
    - flushing and reloading the memory cache
    - updating various tables and lists, etc.
  - Context switch overhead is dependent on hardware support.
    - Multiple register sets in UltraSPARC.
    - Advanced memory management techniques may require extra data to be switched with each context.
  - 100s or 1000s of switches/s typically.
Context Switch (3)

- **Linux example**
  - Total 544,037,375 user ticks = 1511 hours = 63.0 days
  - Total 930,566,190 context switches
  - Roughly 86 context switches / sec (per CPU)
### Process State Queues (1)

- **State queues**
  - The OS maintains a collection of queues that represent the state of all processes in the system
    - Job queue
    - Ready queue
    - Wait queue(s): there may be many wait queues, one for each type of wait (device, timer, message, ...)
  - Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is migrated between the various queues.
Process State Queues (2)

![Diagram showing process state queues for different devices and PCBs.](image)
**Process State Queues (3)**

- **PCBs and state queues**
  - PCBs are data structures
    - dynamically allocated inside OS memory
  - When a process is created:
    - OS allocates a PCB for it
    - OS initializes PCB
    - OS puts PCB on the correct queue
  - As a process computes:
    - OS moves its PCB from queue to queue
  - When a process is terminated:
    - OS deallocates its PCB
Process Creation: UNIX (1)

int fork()

- fork()
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent.
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue.
  - Returns the child’s PID to the parent, and zero to the child.
Process Creation: UNIX (2)

- `exec()`
  - Stops the current process
  - Loads the program “prog” into the process’ address space.
  - Initializes hardware context and args for the new program.
  - Places the PCB on the ready queue.
    - Note: `exec()` does not create a new process.
  - What does it mean for `exec()` to return?
Process Creation: NT

BOOL CreateProcess (char *prog, char *args, ...)

- **CreateProcess()**
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by "prog" into the address space
  - Copies "args" into memory allocated in address space
  - Initializes the hardware context to start execution at main
  - Places the PCB on the ready queue
Why fork()?

- Very useful when the child...
  - is cooperating with the parent.
  - relies upon the parent’s data to accomplish its task.
  - Example: Web server

```c
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```
Inter-Process Communications

- Inside a machine
  - Pipe
  - FIFO
  - Shared memory
  - Sockets

- Across machines
  - Sockets
  - RPCs (Remote Procedure Calls)
  - Java RMI (Remote Method Invocation)