Processes

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

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems
Process Concept

- **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 in Memory

- Kernel virtual memory: "code, data, heap, stack"
- Unused memory: invisible to user code
- User stack: created at runtime
- Run-time heap: managed by malloc
- Read-only segment: ".init, .text, .rodata"
- Read/write segment: ".data, .bss"
- Stack pointer
- BRK

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
Diagram of Process State

- new
- admitted
- interrupt
- exit
- terminated
- ready
- running
- waiting

- I/O or event completion
- scheduler dispatch
- I/O or event wait
### Process State – 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

**l:** Multi-threaded

<table>
<thead>
<tr>
<th>Process ID</th>
<th>State</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>14015</td>
<td>?</td>
<td>[xfslogd] Programming Practices/References/</td>
</tr>
<tr>
<td>14018</td>
<td>?</td>
<td>[jfsI0] Syllabus Work/Thumbs.db</td>
</tr>
<tr>
<td>14019</td>
<td>S</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14020</td>
<td>S</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14021</td>
<td>S</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14022</td>
<td>S</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14023</td>
<td>S</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14024</td>
<td>F</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14025</td>
<td>G</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14026</td>
<td>G</td>
<td>[jfsCommit]</td>
</tr>
<tr>
<td>14027</td>
<td>S</td>
<td>[jfsSync]</td>
</tr>
<tr>
<td>14046</td>
<td>S&lt;</td>
<td>[kworker/0:2]</td>
</tr>
<tr>
<td>17591</td>
<td>S</td>
<td>[kworker/7:0]</td>
</tr>
<tr>
<td>17594</td>
<td>S</td>
<td>[kworker/10:2]</td>
</tr>
<tr>
<td>19063</td>
<td>S&lt;</td>
<td>/usr/bin/python3 /usr/bin/update-manager --no-update --no-focus-on-map</td>
</tr>
<tr>
<td>22037</td>
<td>S</td>
<td>[kworker/2:2]</td>
</tr>
<tr>
<td>22038</td>
<td>S</td>
<td>[kworker/3:0]</td>
</tr>
<tr>
<td>28477</td>
<td>S</td>
<td>[kworker/16:0]</td>
</tr>
<tr>
<td>28799</td>
<td>S</td>
<td>/usr/sbin/cupsd -f</td>
</tr>
<tr>
<td>28802</td>
<td>S</td>
<td>/usr/lib/cups/notifier/dbus dbus://</td>
</tr>
<tr>
<td>29688</td>
<td>S</td>
<td>[kworker16:2]</td>
</tr>
<tr>
<td>29702</td>
<td>S</td>
<td>sshd: jinkyu [priv]</td>
</tr>
<tr>
<td>29846</td>
<td>S</td>
<td>sshd: jinkyu@pts/1</td>
</tr>
<tr>
<td>29850</td>
<td>Ss</td>
<td>-bash</td>
</tr>
<tr>
<td>29876</td>
<td>Ss</td>
<td>ps ax</td>
</tr>
</tbody>
</table>

*Example output from `ps ax` command*
Process Control Block (PCB)

- PCB (Process Control Block)
  - Each PCB represents a process
  - Also called **task control block**
  - 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
    - 2584 bytes as of Linux 4.1.4
### Process Control Block (PCB)

<table>
<thead>
<tr>
<th><strong>Process management</strong></th>
<th><strong>Memory management</strong></th>
<th><strong>File management</strong></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>
Process Representation in Linux

Represented by the C structure `task_struct`

```c
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 */
```
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.
CPU Switch From Process to Process

- process $P_0$
  - executing
  - interrupt or system call
    - save state into PCB$_0$
      - ...
      - ...
    - reload state from PCB$_1$
  - idle

- operating system
- process $P_1$
  - idle
  - executing
    - interrupt or system call
      - save state into PCB$_1$
      - ...
      - ...
    - reload state from PCB$_0$
Context Switch

- 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.
  - Context-switch time is overhead
    - The system does no useful work while switching
    - The more complex the OS and the PCB ➔ the longer the context switch
  - 100s or 1000s of switches/s typically.
Context Switch – Linux Example

- **Linux example**
  - Total 1,693,515,228 ticks = 4704 hours = 196 days
  - Total 4,066,419,922 context switches
  - Roughly 240 context switches / sec
Threads

- So far, a process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple PCs and SPs in PCB
- See next chapter
State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
  - **Job queue** — all processes in the system
  - **Ready queue** — all processes ready to execute
  - **Device queues** — processes waiting for an I/O device (e.g., disk, timer, …)

- 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)
Queueing diagram represents queues, resources, flows
Process State Queues (4)

- 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
Operations on Processes

- System must provide mechanisms for:
  - Process creation,
  - Process termination,
  - and so on as detailed next
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

```
$ cat file1 | 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 Creation (4)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

- 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
Process Termination (1)

- 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
Process Termination (2)

- Process executes last statement and then asks the kernel to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process’ resources are deallocated by the kernel

- Parent process may wait for termination of a child process
  - `pid = wait(&status);`
  - Returns `pid` and termination status of a child

- If no parent waiting (did not invoke `wait()`)
  - Process is a zombie

- If parent terminated without invoking `wait`
  - Process is an orphan
fork()

```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 Creation: UNIX (1)

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?

```c
int exec (char *prog, char *argv[])
```
Process Creation: NT

```c
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

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes
- 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

(a) Message passing.  (b) shared memory.

(a) process A
process B
message queue
m₀ m₁ m₂ m₃ ... mₙ
kernel

(b) process A
shared memory
process B
kernel
Inter-Process Communications

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

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