Lecture 4 – Processes

Recap – Last Lecture

- Operating System Services
- System Calls and Interrupts
- System Programs
- Operating System Design and Implementation
- System Layering
- Virtual Machines
Overview

- Process Concepts
- Process Scheduling
- Operations on Processes
- Inter-process Communication

Process Concept

- Job/Process/Task used interchangeably
- A process is an instance of a program ("program in execution")
- Program: “piece of code”
- Process: code + data + more data + control structures + ...
Process Memory Layout

- **Stack**: Temporary Data
  - Function parameters, local variables, function call frames.
- **Heap**: Dynamically allocated memory.
- **Data**: Global variables & constant strings.
- **Text**: Program code.

Process “Life Cycle”

- NEW
- READY
- RUNNING
- WAITING
- TERMINATED

- Admitted
- Interrupt
- Scheduler dispatch
- I/O or event completion
- Exit
- I/O or event wait
Process State

• A process changes **state** constantly:
  – **New**: being created.
  – **Running**: running on a processor core.
  – **Waiting**: waiting for an *event*.
  – **Ready**: (or **runnable**) waiting for a core.
  – **Terminated**: finished/dead.

OS Information about Processes

• Memory (Stack, Heap, Code, Static Data)
• Current State (e.g., program counter)
• Process ID
• Saved Registers
• Open Files
• Other bookkeeping data

This is called the **process control block (PCB)** or **task control block (TCB)**.
Process Control Block

- process state
- process number
- program counter
- registers
- memory limits
- list of open files
  ...
Process Switching ("Context Switch")

- **Job Queue** is all processes on the system.
- **Ready Queue (Run Queue)** contains processes waiting to run, or **waiting for the CPU**!
- **Device Queue** processes waiting for a device event ("blocked" devices).
- **“Other” Queues** contain processes waiting for other processes to finish, sleeping for time, etc.
Process “Wait” Queues

Schedulers

- **Long-term scheduler** which processes should be run in the future?
  - Degree of multiprogramming!
- **Short-term scheduler** which process should be run next?
  - Manages queues and quickly decides next process to run.
Scheduling Concerns

- Is enough RAM available to satisfy running processes?
- Is device throughput able to support more IO-bound processes?
- Is there enough CPU time available to satisfy all processes? (long) How do I schedule fairly? (short)
- Are there benefits for sleeping (swapping) a process for an extended time? (long)

Schedulers

- Short-term: invoked frequently (milliseconds); must be fast
- Long-term: infrequently (seconds)
- I/O-bound process: spends more time doing I/O than processing (CPU bursts can be frequent, but are short)
- CPU-bound: spends more time doing computations (very long CPU bursts)
Process Creation

- A process is always created via a parent, except for process 1, `/sbin/init`.
- A parent can have multiple children. Entire structure is a tree.
- Each process has a unique identifier, the **process identifier**, or **pid** (get the pid of a process using the getpid() system call).
PSTREE(1)

NAME
pstrace - display a tree of processes

SYNOPSIS
pstrace -V|--version

DESCRIPTION
pstrace shows running processes as a tree. The tree is rooted at either PID or init if PID is omitted. If a user name is specified, all process trees rooted at processes owned by that user are shown.

pstrace visually merges identical branches by putting them in square brackets and prefixing them with the repetition count, e.g.:

init--|getty
    |  |getty
    |-getty
    | | -getty
becomes

init--4[getty]

Child threads of a process are found under the parent process.

The man page pstrace(1) can be viewed by pressing 'h' for help or 'q' to quit.

[Example output]

[pdorne]# ccl2 ->$ pstrace | head -n 20
Init--NetworkManager--chlient
    | (NetworkManager)
    | | -abrt-dump-oops
    | | -abrt
    | | -acpid
    | | -std
    | | -auditd--auditd--auditd--auditd
    | | | | (auditd)
    | | | -autonum--4*{autonum}
    | | | -avahi-daemon--avahi-daemon
    | | | -cachefiles
    | | -cimserver--cimservermain--4*{cimservermain}
    | | -condor_master--chirp_server
    | | | | --condor_source
    | | | | -condor_sched
    | | | -condor_startd--3*{condor_starter--condor_exec.exe--work_queue_work}
    | | | -cron
    | | | -upsd
    | | -dbus-daemon
[pdorne]# ccl2 ->$
Process Creation

- **Resource sharing**
  1. Parent and children share all resources
  2. Children share subset of parent’s resources
  3. Parent and child share no resources

- **Execution**
  1. Parent and children execute concurrently
  2. Parent waits until children terminate

- **Address space**
  1. Child is duplicate of parent
  2. Child has a program loaded into it
### Process: Life Cycle

1. **Parent forks to create a new process**

2. **Child performs actions, possibly exec to run another program**

3. **Parent waits for child process**

4. **Child exits**

5. **Parent receives child’s exit status**

![Diagram showing the process life cycle](image)

### Example Code

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

int main()
{
    pid_t pid;
    /* Fork another 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 */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```
Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call
  - Returns status data from child to parent (via `wait()`)
  - Process’ resources are deallocated by operating system
- Terminate with `abort()`: SIGABRT signal, can be caught; core dump
- Terminate a process from another process: `kill()`: SIGKILL (also “kill -9 pid” from terminal)
- Parent may terminate the execution of children for various reasons:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Signals

Signals are a means of asynchronously notifying a process of an event.

- Each signal delivers a small integer that represents a particular event
- To deliver the event, the kernel will interrupt the normal execution of the process
- Processes register handlers to catch certain events
- After the handlers are executed, the process will continue executing where it was interrupted
Signal: “Kill”

**Kill**

Send a *signal* to a *process*

```c
kill(pid, SIGTERM);
```

**Signal**

Register a callback function for particular event

```c
void handler(int signum) {
    puts("Handler");
}

signal(SIGTERM, handler);
```

SIGCHLD / SIGALRM

**SIGCHLD**

When a *child* process exits, the *parent* is notified with the *SIGCHLD* event

```c
signal(SIGCHLD, handler);
```

**SIGALRM**

An alarm or timer can be set by first using *alarm*, and then handling the *SIGALRM* event when it is triggered

```c
signal(SIGALRM, handler);
alarm(5);
```
Process Termination

- What happens if a process “dies”?
  - The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process:
    ```c
    pid = wait(&status);
    ```
  - Parent “collects” child’s resources
  - If no parent waiting (did not invoke `wait()`) process is a zombie
- What happens if parent “dies”?
  - If parent terminated without invoking `wait`, process is an orphan
  - May receive “new parent” (grandparent, init process, etc.)
  - Cascading termination: no orphans allowed; if a process terminates, all its children must also be terminated

Interprocess Communication

- Processes communicate by sharing data.
- Why do processes communicate?
  - Computation speedup
  - Modularity
  - Information sharing
- Mechanism: interprocess communication (IPC)
- Two standard models: Shared Memory and Message Passing
Communication Models

(a) (b) Shared Memory

Producer-Consumer Problem

• One process produces data. The second process consumes the data.
• Data stored in a buffer:
  – Unbounded-Buffer has no limit on size. Grows to size of memory.
  – Bounded-Buffer has fixed size. Creates a new problem:
    • How do we handle the producer creating data too fast?
Shared Memory Solution

Circular buffer

\[
\begin{array}{cccccccccc}
\text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} \\
\text{IN} & \text{OUT} \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{A} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} & \text{NUL} \\
\text{OUT} & \text{IN} \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{NUL} & \text{NUL} & \text{C} & \text{D} & \text{E} & \text{F} & \text{NUL} & \text{NUL} & \text{NUL} \\
\text{OUT} & \text{IN} \\
\end{array}
\]

• Shared data

```c
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

• Solution is correct, but can only use BUFFER_SIZE-1 elements
Bounded Buffer - Producer

```
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;
}
```

Bounded Buffer - Consumer

```
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 */
}
```
Message Passing

- Two primitives:
  - \texttt{send} (C, message) – messages have maximum size
  - \texttt{receive} (P, message)
- Think \textit{mailboxes}.
- \textbf{Kernel} usually manages the message passing and “mailboxes”.

Message Passing Considerations

- How is the link established?
  - Automatically on send?
- Can the link be asymmetric?
  - Receiving a message: who is the sender?
- Is there a limit to the capacity of the link?
- Is the message size fixed or variable?
- Is a link unidirectional or bidirectional?
- Can there be multiple links between a pair of communication processes?
Message Passing

• Direct Communication
  – send (P, message) -> receiver process P
  – receive (Q, message) -> sender process Q

• Indirect Communication ("mailboxes")
  – send (M1, message) -> put in mailbox M1
  – receive (M1, message) -> take from mailbox M1

IPC Synchronization

• Blocking?
  – Consumer is put in a waiting scheduler queue if "mailbox" is empty.
  – Producer is put in a waiting scheduler queue if "mailbox" is full.

• Non-blocking?
  – Neither Producer nor Consumer blocks; failure is returned from message passing primitive instead.
Buffering

- Queue of messages attached to the link; implemented in one of three ways:
  - **Zero capacity** – no messages are queued on a link. 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.

IPC - POSIX

- POSIX Shared Memory
  ```c
  shm_id = shm_open(name, O_CREAT | O_RDWR, 0666);
  ftruncate(shm_id, 4096);
  shared_memory = (char *) shmat(shm_id, NULL, 0);
  sprintf(shared_memory, "Writing to shared memory");
  Also: shmdt (remove), shmctl (destroy)
  ```
IPC - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation: Kernel and Notify
  - Only three system calls needed for message transfer
    - `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via `port_allocate()`

Recap

- Key Points:
  - System Layering
  - Concept of a Process
  - Scheduling
  - Process Creation and Termination
  - Interprocess Communication