FAQ

- **API vs system call** User programs in a high level language use APIs, APIs are wrappers for system calls that call system routines. [Example Linux x-86 system call code.](#)

- **System call examples: soon in the class**

- **Why do we need API (application programing interface)?** So that we don’t have to write the code in assembly. [Example](#)

- **Who came up with API standard POSIX?** Committees of experts.

- **Is Windows POSIX compliant?** Yes, ..
Course notes

• HW2 available due Feb 10.
  – Help Session 2: This wed 5:30 PM Room CSB 425
• Self exercises in Teams
• TA: Office hours course website/TEAMSs
  – Available on Help Desk in TEAMs
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
ICQ 9/7/21
CS370 OS  Ch3  Processes

• Process Concept: a program in execution
• Process Scheduling
• Processes creation and termination
• Interprocess Communication using shared memory and message passing
• An operating system executes a variety of programs:
  – Batch system – jobs
  – Time-shared systems – user programs or tasks
• Textbook uses the terms job and process almost interchangeably
• Process – a program in execution; process execution must progress in sequential fashion. Includes
  – The program code, also called “text section”
  – Current activity including program counter, processor registers
  – Stack containing temporary data
    • Function parameters, return addresses, local variables
  – Data section containing global variables
  – Heap containing memory dynamically allocated during run time
Transitions:

**Ready to Running**: scheduled by scheduler

**Running to Ready**: scheduler picks another process, back in ready queue

**Running to Waiting** (Blocked): process blocks for input/output

**Waiting to Ready**: I/O or event done
Process Control Block (PCB)

Information associated with each process (also called task control block)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files
CPU Switch From Process to Process

Diagram:

- Process $P_0$ executing
- Operating system
- Process $P_1$

Actions:
- Interrupt or system call
  - Save state into PCB$_0$
  - Reload state from PCB$_1$
  - Save state into PCB$_1$
  - Reload state from PCB$_0$

States:
- Idle
- Executing

Flow:
- Process $P_0$ executing
- Operating system
- Process $P_1$ executing
- Operating system
- Process $P_0$ idle
- Process $P_1$ idle
Processes on my computer

• Mac: apps > utilities > activity monitor > CPU etc.
    – See information about processes
    – Name, PID, threads, details ..

• Windows 10  Ctrl+Alt+Del
Threads

- So far, 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 program counters in PCB
- Coming up in next chapter
Represented by the C structure `task_struct`.

Fields may include:

```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 */
```

Unlike an array, the elements of a struct can be of different data types.
Process Scheduling
Process Scheduling

• Maximize CPU use, quickly switch processes onto CPU for time sharing
• **Process scheduler** selects among available processes for next execution on CPU
• Maintains **scheduling queues** of processes
  – **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
Ready Queue And Various I/O Device Queues
Queueing diagram represents queues, resources, flows

Assumes a single CPU. Common until recently
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **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
- Long-term scheduler strives for good **process mix**
Multitasking in Mobile Systems

• Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended.

• Due to screen real estate, user interface limits iOS provides for a
  – Single **foreground** process- controlled via user interface
  – Multiple **background** processes– in memory, running, but not on the display, and with limits
    • Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

• Newer iOS supports multitasking better. iOS 14: picture in picture

• Android runs foreground and background, with fewer limits
  – Background process uses a **service** to perform tasks
  – Service can keep running even if background process is suspended
  – Service has no user interface, small memory use.
ARE YOU SURE YOU WANT TO SIT AT THE KIDS TABLE?
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**

- **Context** of a process represented in the PCB

- 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

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
Processes creation & termination
Process Creation

• **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

• Generally, process identified and managed via a **process identifier (pid)**

• Resource sharing options
  – Parent and children share all resources
  – Children share subset of parent’s resources
  – Parent and child share no resources*

• Execution options
  – Parent and children execute concurrently
  – Parent waits until children terminate
A Tree of Processes in Linux

init
  pid = 1

login
  pid = 8415

bash
  pid = 8416

ps
  pid = 9298

emacs
  pid = 9204

kthreadd
  pid = 2

khelper
  pid = 6

pdflush
  pid = 200

ssh
  pid = 3028

sshd
  pid = 3610

tcsch
  pid = 4005

bash
  pid = 9216

ps
  pid = 9298
• 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
Fork ( ) to create a child process

- Fork creates a copy of process
- Return value from fork (): integer
  - When > 0:
    - Running in (original) Parent process
    - return value is pid of new child
  - When = 0:
    - Running in new Child process
  - When < 0:
    - Error! Perhaps exceeds resource constraints. sets errno (a global variable in errno.h)
    - Running in original process

- All of the state of original process duplicated in both Parent and Child! Almost ..
  - Memory, File Descriptors (next topic), etc...
Process Management System Calls

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process. Several variations.
- UNIX wait – system call to wait for a process to finish
- Details: see man pages

Some examples:

```c
pid_t pid = getpid();  /* get current processes PID */;
waitpid(cid, 0, 0);  /* Wait for my child to terminate. */
exit (0);  /* Quit*/
kill(cid, SIGKILL);  /* Kill child*/
```
UNIX Process Management

```c
int pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);

main () {
...
}
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
Forking PIDs

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

int main()
{
    pid_t cid;

    /* fork a child process */
    cid = fork();
    if (cid < 0) /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (cid == 0) /* child process */
        printf("I am the child %d, my PID is %d\n", cid, getpid());
        execlp("/bin/ls","ls",NULL);
    }
    else /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent with PID %d, my parent is %d, my child is %d\n",getpid(), getppid(), cid);
        wait(NULL);

        printf("Child Complete\n");

    return 0;
}
```
wait/waitpid

• Wait/waitpid ( ) allows caller to suspend execution until child’s status is available
• Process status availability
  – Generally after termination
  – Or if process is stopped
• pid_t waitpid(pid_t pid, int *status, int options);
• The value of pid can be:
  – 0 wait for any child process with same process group ID (perhaps inherited)
  – > 0 wait for child whose process group ID is equal to the value of pid
  – -1 wait for any child process (equi to wait ( ))
• Status: where status info needs to be saved
Linux: fork ( )

- Search for man fork( )

**NAME** fork - create a child process

**SYNOPSIS**
```
#include <unistd.h>

pid_t fork(void);
```

**DESCRIPTION** fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. …

The child process and the parent process run in separate memory spaces…

The child process is an exact duplicate of the parent process except for the following points: ….

**RETURN VALUE** On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

**EXAMPLE** See pipe(2) and wait(2).

…

errno is a global variable in errno.h
Process Group ID

• Process group is a collection of related processes
• Each process has a process group ID
• Process group leader?
  – Process with pid==pgid
• A process group has an associated controlling terminal, usually the user’s keyboard
  – Control-C: sends interrupt signal (SIGINT) to all processes in the process group
  – Control-Z: sends the suspend signal (SIGSTOP) to all processes in the process group

Applies to foreground processes: those interacting with the terminal
A child inherits parent’s process group ID. Parent or child can change group ID of child by using setpgid.

By default, a Process Group comprises:
• Parent (and further ancestors)
• Siblings
• Children (and further descendants)

A process can only send signals to members of its process group
• Signals are a limited form of inter-process communication used in Unix.
• Signals can be sent using system call
  – int kill(pid_t pid, int sig);
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

• Parent may terminate the execution of children processes using the `kill()` system call. Some reasons for doing so:
  – 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

```c
kill(child_pid, SIGKILL);
```
Process Termination

• Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  – **cascading termination.** All children, grandchildren, etc. are terminated.
  – The termination is initiated by the operating system.
• 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
  \[ \text{pid} = \text{wait}(&\text{status}); \]
• If no parent waiting (did not invoke `wait()`) process is a **zombie**
• If parent terminated without invoking `wait`, process is an orphan (it is still running, reclaimed by init)

Zombie: a process that has completed execution (via the exit system call) but still has an entry in the process table
Multi-process Program Ex – Chrome Browser

• Early web browsers ran as single process
  – If one web site causes trouble, entire browser can hang or crash

• Google Chrome Browser is multiprocess with 3 different types of processes:
  – **Browser** process manages user interface, disk and network I/O
  – **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    • Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  – **Plug-in** process for each type of plug-in

![Chrome browser tab example with each tab representing a separate process.](image-url)
Multitasking
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
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing
Producer-Consumer Problem

• Common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  
  – **unbounded-buffer** places no practical limit on the size of the buffer
  
  – **bounded-buffer** assumes that there is a fixed buffer size

---

**Why do we need a buffer (shared memory region)?**
- The producer and the consumer process operate at their own speeds. Items wait in the buffer when consumer is slow.
- Where does the bounded buffer “start”
  - It is circular
Bounded-Buffer – Shared-Memory Solution

- **Shared data**
  ```c
  #define BUFFER_SIZE 8
  typedef struct {
      ...}
  item;

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

- **in** points to the **next free position** in the buffer
- **out** points to the **first full position** in the buffer.
- Buffer is empty when **in == out**
- Buffer is full when
  ```
  ((in + 1) % BUFFER_SIZE) == out.  (Circular buffer)
  ```
- This scheme can only use **BUFFER_SIZE-1** elements.

![Circular buffer diagram](image)

\[(2+1)\%8 = 3 \text{ but } (7+1)\%8 = 0\]
item next_produced;
while (true) {
    /* produce an item in next produced */
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
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 */
}

```
0 1 2 3 4 5 6 7
```
Interprocess Communication – Shared Memory

• Each process has its own private address space.
• An area of memory shared among the processes that wish to communicate.
• The communication is under the control of the user processes, not the operating system.
• Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  – Synchronization is discussed in great details in a later Chapter.
• Example soon.

Only one process may access shared memory at a time.
Interprocess Communication – Message Passing

• Mechanism for processes to communicate and to synchronize their actions

• Message system – processes communicate with each other without resorting to shared variables

• IPC facility provides two operations:
  – send(message)
  – receive(message)

• The message size is either fixed or variable
- If processes $P$ and $Q$ wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be 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?
• Implementation of communication link
  
  – Physical:
    • Shared memory
    • Hardware bus
    • Network
  
  – Logical: Options (details next)
    • Direct (process to process) or indirect (mail box)
    • Synchronous (blocking) or asynchronous (non-blocking)
    • Automatic or explicit buffering