CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2017 Lecture a

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
FAQ

- ?

Notes: TA office hours on the web
PA1 available. Reward for submitting a week earlier. HS1 slides/recording.
Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
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

- 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: Input available
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

- **Process $P_0$**: executing
- **Operating System**: interrupt or system call
  - save state into PCB$_0$
  - ...
  - ...
  - reload state from PCB$_1$
- **Process $P_1$**: idle
  - executing
  - interrupt or system call
  - save state into PCB$_1$
  - ...
  - ...
  - reload state from PCB$_0$
- **Operating System**: idle
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
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 */
```
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
Representation of Process Scheduling

- **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**
Medium-term scheduler can be added if degree of multiple programming needs to decrease

- Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**
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

• 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
FAQ

• Program control Block – What does it include and why? Where is it saved? another look

• Process Scheduling – Why? How? more soon

• How do multiple processes, that are part of a single program, interact. Inter-process communication details coming up.

• What happens to a process once it is finished? Resources deallocated, but only after ..

• If a process forks a child, what happens to the parent? It continues.

• When the CPU is running user-processes how does the OS run? Is the kernel one process? Where does it reside?

• Can a process induce its own context switch? Yes, we’ll see how.
Notes

• Please register your iClicker: Canvas > iClicker menu item
  – Bring to class
• Introduce yourself on Canvas
• Canvas Discussion board available for discussions
  – usual rules apply
• The TA office hours and photos are on cs270 home page

More FAQ

• Q3.1 Registers are managed by the (a) Compiler
• Q2.b: Multiprogramming requires presence of multiple processors. False
• Q4.b POSIX API are used in a high level language. True
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
• 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!
  - 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

Notes:

```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

fork

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

exec

```c
main () {
    ...
}
```

wait

```c
pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);
```
```c
#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 */
        execlp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

`execlp(3) - Linux man page http://linux.die.net/man/3/execlp`
Forking PIDs

```c
#include <stdlib.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;
}
```

Ys-MacBook-Air:ch3 ymalaiya$ ./newproc-posix_m
I am the parent with PID 494, my parent is 485, my child is 496
I am the child 0, my PID is 496
DateClient.java
```
newproc-posix_m
Child Complete
Ys-MacBook-Air:ch3 ymalaiya$
```
• 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
  – Others
• Status: where status info needs to be saved
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 `abort()` 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
Some operating systems do not allow child to exists 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:

```c
pid = wait(&status);
```

- If no parent waiting (did not invoke `wait()`) process is a zombie
- If parent terminated without invoking `wait`, process is an orphan
Multiprocess Architecture – 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
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*
Communications Models

(a) Message passing. (b) shared memory.
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
Bounded-Buffer – Shared-Memory Solution

• Shared data
  
  ```c
  #define BUFFER_SIZE 10
  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
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;
}
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 */
}

Out

0 1 2 3 4 5 6 7

In
Interprocess Communication – Shared Memory

- 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 issues 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 Chapter 5.
- 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
Message Passing (Cont.)

- 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
Direct Communication

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

• Messages are 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

• Operations
  – create a new mailbox (port)
  – send and receive messages through mailbox
  – destroy a mailbox

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

• Mailbox sharing
  – $P_1$, $P_2$, and $P_3$ share mailbox A
  – $P_1$, sends; $P_2$ and $P_3$ receive
  – Who gets the message?

• Possible 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.