CS 471 Operating Systems

Yue Cheng

George Mason University
Fall 2017
Outline

- Process concept
- Process creation
- Process states and scheduling
- Preemption and context switch
- Inter-process communication
Process Concept

- **Process**: a *program in execution*
  - Process execution must progress in sequential fashion

- A *program* is a passive entity, whereas a process is an *active entity* with a program counter and a set of associated resources

- Each process has its own *address space*:
  - *Text section (text segment)* contains the executable code
  - *Data section (data segment)* contains the global variables
  - *Stack* contains temporary data (local variables, return addresses..)
  - A process may contain a *heap*, which contains memory that is dynamically allocated at run-time

✓ The program counter and CPU registers are part of the *process context*
Process

- Introduced to obtain a systematic way of monitoring and controlling program execution
- At first:
  - The unit that can be dispatched (scheduled)
  - The unit that ‘owns’ resources
  (This view changed later on with the advent of threads…)
- A process is an executable program with:
  - associated data (variables, buffers…)
  - execution context
Processes

a) Multiprogramming of four programs
b) A conceptual model of 4 independent, sequential processes
c) Only one program active at any instant
OS Requirements for Processes

- **Utilization**
  - OS must *interleave* the execution of several processes to maximize CPU utilization while providing reasonable response time

- **Synchronization**
  - OS must allocate resources to processes while *avoiding* deadlock

- **Communication**
  - OS must support *inter-process communication* and user creation of processes
Process Creation

- Principle events that cause process creation
  - System initialization
  - Execution of a process creation system call by a running process
  - User request to create a process
Process Creation (cont.)

- Parent process creates children processes, which, in turn create other processes, forming a tree (hierarchy) of processes

- Issues
  - Will the parent and child execute concurrently?
  - How will the address space of the child be related to that of the parent?
  - Will the parent and child share some resources?
An Example Process Tree

```
      Sched
        | pid = 0
      ┌───┐
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      │   │
      └───┘
        https://example.com
```
How to View Process Tree in Linux?

- % ps auxf
  - f is the option to show the process tree

- % pstree
Process Creation in Linux

- Each process has a process identifier (pid)
- The parent executes fork() system call to spawn a child
- The child process has a separate copy of the parent’s address space
- Both the parent and the child continue execution at the instruction following the fork() system call. The return code for the fork() system call is
  - zero for the new (child) process
  - the (nonzero) pid for the parent
- Typically, a process can execute a system call like execlp() to load a binary file into memory
Example Program with “fork”

```c
void main ()
{
    int pid;

    pid = fork();
    if (pid < 0) {error_msg}
    else if (pid == 0) {
        /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else {
        /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        exit(0);
    }
}
```
A Very Simple Shell

while (1) {
  
  type_prompt();
  read_command(cmd);
  pid = fork();
  if (pid < 0) {error_msg}
  else if (pid == 0) { /* child process */
    execute_command(cmd);
  }
  else { /* parent process */
    wait(NULL);
  }
}
What happens to the value of number?

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

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        printf("\tIn the child, the number is %d — PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In the parent, the number is %d\n", number);
    }

    return 0;
}
```
Results

./forkexample1

Running the fork example
The initial value of number is 7
PID is 2137
PID is 0

   In the child, the number is 49  PID is 0

In the parent, the number is 7
Demo: fork 2

What happens to the value of number?

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

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the fork example\n");
    printf("The initial value of number is %d\n", number);

    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        number *= number;
        fork();
        printf("In the child, the number is %d -- PID is %d\n", number, pid);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In the parent, the number is %d\n", number);
    }

    return 0;
}
```
Results

./forkexample2

Running the fork example
The initial value of number is 7
PID is 2164
PID is 0
In the child, the number is 49  PID is 0
In the child, the number is 49  PID is 0
In the parent, the number is 7
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int number = 7;

int main(void) {
    pid_t pid;
    printf("\nRunning the execl example\n");
    pid = fork();
    printf("PID is %d\n", pid);

    if (pid == 0) {
        printf("In the execl child, PID is %d\n", pid);
        execl("./forkexample2", "forkexample2", NULL);
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("In the parent, done waiting\n");
    }

    return 0;
}
Results

./execlexample
Running execl code
PID is 2179
PID is 0

          In the execl child,  PID is 0

Running the fork example
The initial value of number is 7
PID is 2180
PID is 0

          In the child, the number is 49  PID is 0
          In the child, the number is 49  PID is 0
In the parent, the number is 7
In the parent, done waiting
Process Termination

- Process executes **last statement** and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait or waitpid)
  - Process’ resources are de-allocated by operating system

- **Parent** may terminate execution of children processes (e.g. TerminateProcess() in Win32)

- Process may also terminate due to **errors**

- **Cascading** termination – when a system does not allow a child process to continue after the parent has terminated
Reasons for Process Termination

- Normal completion
- Memory unavailable
- Memory bounds violation
- Protection error
  - E.g., write to read-only file
- Arithmetic error
- Time limit exceeded
- Time overrun
  - Process waited longer than a specified maximum for an event
- And a lot more…
Reasons for Process Termination (cont.)

- I/O failure
- Invalid instruction
  - happens when try to execute data
- Privileged instruction
- Operating system intervention
  - such as when deadlock occurs
- Parent request to terminate one offspring
- Parent terminates so child processes terminate
Process States (simplified)

- **Running** state
- **Ready** state
- **Blocked** state
- **New** state
  - OS has performed the necessary actions to *create* the process but has not yet admitted the process
- **Exit** state
  - Termination moves the process to this state
  - Tables and other info are temporarily preserved for auxiliary program
A Five-State Process Model

Ready to exit: A parent may terminate a child process
Process Queues

Admit → Ready Queue → Dispatch → Processor → Release

Event 1 Occurs → Event 1 Queue → Event 1 Wait

Event 2 Occurs → Event 2 Queue → Event 2 Wait

Event n Occurs → Event n Queue → Event n Wait

Timeout
Swapping/Suspending

- Processes may need to be *swapped* out to disk
  - This is true with virtual memory!

- 2 new states:
  - **Blocked Suspend**: blocked processes which have been swapped out to disk
  - **Ready Suspend**: ready processes which have been swapped out to disk
Additional State Transitions

- **Blocked --> Blocked Suspend**
  - When all processes are blocked, OS will make room to bring a ready process in memory

- **Blocked Suspend --> Ready Suspend**
  - When the event for which it has been waiting occurs

- **Ready Suspend --> Ready**
  - When no more ready process in main memory

- **Ready---> Ready Suspend**
  - When there are no blocked processes and must free memory for adequate performance
A Seven-State Process Model
Process Scheduling

- The operating system is responsible for managing the *scheduling* activities
  - A uniprocessor system can have only one running process at a time
  - The main memory cannot always accommodate all processes at runtime for multiprogrammed OS
  - The operating system will need to decide on which process to execute next (*CPU scheduling*), and which processes will be brought to the main memory (*job scheduling*)
Process Scheduling (cont.)

- How to pick which process to run?
- Scan process table for first runnable?
  - Expensive: Weird priorities (e.g., small pids do better)
  - Divide into runnable and blocked processes
- FIFO (FCFS)?
  - Put processes on back of list, pull them from front

```
head <-> t1 <-> t2 <-> t3 <-> t4 <-> tail
```

- Give some processes a better shot at the CPU?
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting for CPU
- **Device queues** – set of processes waiting for an I/O device
- Process migration is possible among these queues
CPU and I/O Bursts

CPU–I/O Burst Cycle

- Process execution consists of a *burst* of CPU execution and I/O wait.

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.
CPU-bound vs. I/O-bound Processes

(a) A CPU-bound process
(b) An I/O-bound process
Scheduling Policies

- **Want to balance multiple goals**
  - Fairness – don’t starve processes
  - Priority – reflect relative importance of jobs
  - Deadlines – must do X (play video) by certain time
  - Throughput – want good overall performance
  - Efficiency – minimize overhead of scheduler itself

- **No universal policy**
  - Many variables, multiple objectives, can’t optimize them all
  - Conflicting goals (e.g., throughput vs. priority)

- **We will spend a whole lecture on this topic**
Preemption

- Can preempt a process when kernel gets control
- Running process can vector control to kernel
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- Periodic timer interrupt
  - If running processes used up quantum, schedule another
- Device interrupt
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than currently running process
- Changing running process is called a context switch
Example of Preemption Act

Each square: one CPU cycle
1: Process 1
2: Process 2
Context Switch

process $P_0$ operating system process $P_1$

interrupt or system call

save state into PCB$_0$

interrupt or system call

save state into PCB$_1$

reload state from PCB$_1$

reload state from PCB$_0$

executing

idle

executing

idle
Context Switch (cont.)

- Very machine dependent. Typical things include
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translation

- **Non-negligible runtime cost!**
  - Save/restore floating point registers expensive
    - Optimization: only save if process uses floating points
  - May require flushing TLB (memory address translation hardware)
  - Usually cause more cache misses (switch working set)
Process Communication

- Mechanism for processes to communicate and to synchronize their actions.

- Two models
  - Communication through a shared memory region
  - Communication through message passing
Previously, in a distributed system, message-passing was the only possible communication model. However, remote direct memory access (RDMA) technique bridges this gap by providing remote memory access through network.
Communication through Message Passing

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

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`

- With *indirect communication*, the messages are sent to and received from mailboxes (or, ports)
  - `send (A, message) /* A is a mailbox */`
  - `receive (A, message)`
Communication through Message Passing

- Message passing can be either blocking (synchronous) or non-blocking (asynchronous)
  - **Blocking Send**: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - **Non-blocking Send**: The sending process resumes the operation as soon as the message is received by the kernel
  - **Blocking Receive**: The receiver blocks until the message is available
  - **Non-blocking Receive**: “Receive” operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message
Communication through Shared Memory

- The memory region to be shared must be explicitly defined.
- Using system calls – in Linux:
  - `shmget` creates a shared memory block.
  - `shmat` maps an existing shared memory block into a process’s address space.
  - `shmdt` removes (“unmaps”) a shared memory block from the process’s address space.
  - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …).
- Problems with simultaneous access to the shared variables.
- Compilers for concurrent programming languages can provide direct support when declaring variables (e.g. “shared int buffer”)


Shared Memory Example

#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{

    pid_t pid;
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char *shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int segment_size = 4096;

    /** allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

    /** attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);
    printf("shared memory segment %d attached at address %p\n", segment_id, shared_memory);

    /** write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there CS 471 \n");
/* Do a fork */
    pid = fork();

    if (pid == 0){
        printf("\n\tIn the execl child, PID is %d memory is %p\n", pid, shared_memory);
        execlp("./sharechild.exe", "sharechild.exe", shared_memory, NULL);
        return 0;
    }

    else if (pid > 0) {
        wait(NULL);
        printf("In the parent, done waiting\n");
    }

    /** now print out the string from shared memory */
    printf("\nIn Parent==>\%s\n", shared_memory);

    /** now detach the shared memory segment */
    if ( shmdt(shared_memory) == -1) {
        fprintf(stderr, "Unable to detach\n");
    }

    /** now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);
    return 0;
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{

  /*Print out the string from shared memory */
  printf("\nFrom Child==>%s\n\n", argv[1]);

  return 0;
}
Output

./share.exe
shared memory segment 720896 attached at address 0x10b74c000

In the execl child,  PID is 0  memory is 0x10b74c000

From Child==>Hi there  CS 471!
*

In the parent, done waiting

In Parent==>Hi there  CS 471!
*