

Operating System Processes vs. Threads

CIT 595
Spring 2008

Process Creation

- A process can create several other process a.k.a child or sub processes
 - Exploit the ability of the system to concurrently execute
 - E.g. **gcc** program invokes different processes for the compiler, assembler, and linker
- Each process could get its resources directly from OS
 - Can restrict resources to subset of parent's process
 - Prevent overloading of system with too many children
- The new process gets its own space in memory
 - Parent and child processes address space are still different
- Because parent and child are isolated, they can communicate only via system calls

CIT 595

2

Process Creation Ability

- OS usually provide a mechanism for process creation and termination via system calls
- Programmers usually access these system calls via application interface (API) / library
- The standard C library provides a set of convenient wrapper functions for most frequently used system calls

```
<unistd.h>
<sys/types.h>
pid_t getpid(void) ; return PID of the caller
pid_t getppid(void); return PID of the caller's parent
```

CIT 595

3

Process Creation in Unix

- Process creation is done using *fork()* provided in `<unistd.h>`
- Child process is exact clone of the parent but with it's own copy

```
int main(){
    printf("Hello\n");
    fork();
    printf("Bye\n");
    return 0;
}
```

- child starts running as soon as `fork()` return
- Hello is printed once
- Bye is printed twice

CIT 595

4

Example of Process Creation using C

```
//includes not shown
int main(){
    pid_t pid;
    int status;
    pid_t s;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        //Overlay address space with UNIX
        command "ls"
        printf("child pid = %d\n", getpid());
        if(exec("/bin/ls", "ls", "-la", NULL) < 0){
            printf("Command not found\n");
            exit(1);
        }
    }
    else { /* parent process */
        /* parent will wait for the child to
        complete */
        printf("#####I am the
        parent#####\n");
        s = wait (&status);
        if(s != -1)
            printf("Parent detect child
            process # %d is done\n", s);
        return 0;
    }
}
exit(0);
}
```

CIT 595

5

Make child do something useful ?

- After fork which process is parent ? child?
pid_t fork(void);
 - For child, fork() return value is 0
 - For parent, fork() return value is the PID of the child
 - fork() returns -1 if error occurred
- To make child do something overload its address space with what you want it do
 - Use *exec()* system call (and its variants)
 - E.g. `int exec(const char *path, const char *arg, ...);`
 - Args to exec are
 - location of the executable code
 - Array of string pointers
 - There are many variations on exec
 - See man pages for `exec()` and `execv`
 - If error occurred then returns -1

CIT 595

6

Parent Waiting on child to complete

- Often a process will have nothing do until its child terminates
 - E.g. what the unix shell usually does when you type a new command
- *pid_t wait(int *)*
 - Blocks parent until child terminates
 - On success, returns the pid of the child that exited or -1 if error occurred
 - Once the child exits, the integer value reference (passed to wait) is filled in with the particular status
 - The parent then can use this status to do something
 - Other variation on wait as well

CIT 595

7

Process Termination

- *exit()* forces the current process to terminate
 - Is also automatically called when the process ends
- OS passes the child's exit status to the parent and then discards the process
- On discarding the process all the resources such as memory and I/O are de-allocated

CIT 595

8

Multithreading

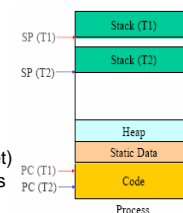
- Multithreading a program *appears* to do more than one thing at a time
- The idea of Multithreading is same as Multiprocessing i.e. multitasking but within a single process
 - Multiprocessing is multi-tasking across different process
- E.g. A word processing program has separate threads:
 - One for displaying graphics
 - Other for reading in keystrokes from the user
 - Another to perform spelling and grammar checking in the background

CIT 595

9

Multithreading (contd..)

- Allow process to be subdivided into different threads of control
- A *thread* is the smallest schedulable unit in multithreading
- A thread in execution works with
 - thread ID
 - Registers (program counter and working register set)
 - Stack (for procedure call parameters, local variables etc.)
- A thread *shares* with other threads a process's (to which it belongs to)
 - Code section
 - Data section (static + heap)

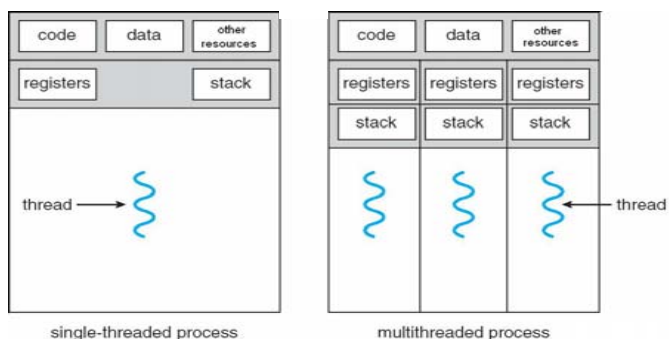


Process with 2 threads

CIT 595

10

Difference between Single vs. Multithread Process



- A process by itself can be viewed a single thread and is traditionally known as a heavy weight process

CIT 595

11

Advantages of Multithreading

- Increase responsiveness to the user
 - Allows a program to continue running even if parts of it is "waiting"
- Resource Sharing
 - Threads share memory and resources of the process to which they belong
 - All threads run within same address space
- Economical
 - They can communicate through shared data and thereby eliminate the overhead of system calls

CIT 595

12

Process and Threads

- Like process states, threads also have states:
 - New, Ready, Running, Waiting and Terminated
- Like processes, the OS will switch between threads (even if though they belong to a single process) for CPU usage
- Like process creation, thread creation is supported by APIs
- Java Threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface
- In C, threads are created using functions in <pthread.h> library

CIT 595

13

Sharing Address Space

- Sharing address space requires only one copy of code or data in main memory
 - E.g.1: 2 processes share the same library routine (code)
 - E.g.2: A print program produces characters and that is consumed by printer driver (two processes sharing data)
 - E.g.3: Threads within a process share (global) data section
- As long as shared data is not being modified there is no problem
- But concurrent access to shared data that modify the value of the data can lead to *data inconsistency*
 - E.g. Printer driver consumed data before print program produced it

CIT 595

14

Threading Example

```
class Counter {
  private int c = 0;
  public void increment() {
    c++;
  }
  public void decrement() {
    c--;
  }
  public int value() {
    return c;
  }
}
```

- If a Counter object is referenced from multiple threads
- There will be interference between threads when 2 operations (increment and decrement), running in different threads, but acting on the same data (i.e. c)
- This means that the two operations consist of multiple steps, and the sequences of steps overlap.

CIT 595

15

Threading Example (contd..)

- Remember that single expression “c++” can be decomposed into three steps:
 1. Retrieve the current value of c.
 2. Increment the retrieved value by 1.
 3. Store the incremented value back in c.
- The same applies for c--

CIT 595

16

Threading Example (contd..)

- Suppose Thread 1 invokes increment at about the same time Thread 2 invokes decrement
 - In reality OS is going to switch between Thread 1 and 2
- If the initial value of c is 0, their interleaved actions might follow this sequence:
 1. Thread 1: Retrieve c
 2. Thread 2: Retrieve c
 3. Thread 1: Increment retrieved value; result is 1
 4. Thread 2: Decrement retrieved value; result is -1
 5. Thread 1: Store result in c; c is now 1
 6. Thread 2: Store result in c; c is now -1

CIT 595

17

Race Condition

- In previous example, Thread 1's result is lost, overwritten by Thread 2
 - Many different interleaving can result in different value of "c"
- A situation where several process/threads access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is known as *Race Condition*

CIT 595

18

Synchronization

- To avoid race conditions, need to guarantee exclusive access to the shared data at any point in time
- Synchronization is achieved by
 - Request entry to critical section (regions of code that may change shared data)
 - Access (shared) data in critical section
 - Request exit from critical section
- Synchronization can be conceptualized as *locking* mechanism
 - When a thread/process is in critical-section, the region is locked for others
 - Only can enter critical section if lock is open

CIT 595

19

Implementing Synchronization

- The lock state is implemented by a memory location
- The location contains value 0 if the lock is unlocked and 1 if the lock is locked
- This is e.g. with two threads/processes can be further expanded to multiple threads/processes
- The update to this memory location is atomic (no interruptions) and is guaranteed by the OS

CIT 595

20

Synchronization Mechanics for Programmer

■ High-Level Language constructs which inherently translates to OS system calls

■ E.g. In **Java** you can synchronize methods using **synchronized** keyword

■ Guarantees mutual exclusion i.e. acquires the intrinsic lock for that method's object and releases it when method returns

■ Guarantees that changes to the state of the object are visible to all threads

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

CIT 595

21

Implementing Threads in C using <pthread.h>

```
void *PrintHello(void * threadid){
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%d!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

CIT 595

22

Thread Creation

int pthread_create (pthread_t * thread, pthread_attr_t *attr, void * (*start_routine) (void *), void * arg)

- **thread**: unique identifier for the new thread returned by the subroutine
- **attr**: attribute object that may be used to set thread attribute
 - You can specify a thread attributes object, or NULL for the default values
 - Attributes include stack location, stack size, priority
- **start_routine**: the C routine that the thread will execute once it is created
 - function that thread perform must be void * funcname (void *)
- **arg**: A single argument that may be passed to **start_routine**.
 - It must be passed by reference as a pointer cast of type void
 - NULL may be used if no argument is to be passed
 - It should be cast to its correct type in the function

■ returns 0 when successful

CIT 595

23

Thread Termination

- To terminate a thread:
void pthread_exit(void *value_ptr);
- **value_ptr** is a pointer to the object (variable, array, structure) returned by the thread
 - The value must not be of local scope otherwise it won't exist after the thread is destroyed
 - Can be NULL if not returning anything
- Note: To compile program
 - gcc filename.c -lpthread
 - Must provide -lpthread flag

CIT 595

24

Thread join

- Causes the calling thread to wait for another thread to terminate
 - Just like wait() for processes
 - Parent process do not have to wait for children
 - For threads it important as we run the risk of executing an exit (reach end of main) which will terminate the process and all threads before the threads have completed
- `int pthread_join(pthread_t thread, void ** value_ptr)`
 - `thread` is the id of the thread to wait on
 - `value_ptr` is the value given to `pthread_exit()` by the terminating thread
 - returns 0 to indicate success

- Usage

```
void * return_val;
....
//after code creating threads
if(pthread_join(worker_thread, &return_val)){
    printf("Error while waiting on thread\n");
    exit(1);
}
```

CIT 595

25

Thread mutex

- Mutex is short for mutual exclusion
- Provides lock mechanism for shared data
- To create a mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t * attr)
```

 - `mutex` is the lock (of type `pthread_mutex_t`)
 - `attr` is the lock attributes
 - NULL by default
- To lock: `int pthread_mutex_lock(pthread_mutex_t *mutex)`
 - If lock is already locked the calling thread is blocked
 - If lock is not locked the calling thread acquires it
 - returns 0 on success
- To Unlock: `pthread_mutex_unlock`

CIT 595

26

Mutex (contd..)

- Declare variables as globals i.e. outside all methods
 - E.g. `pthread_mutex_t myMutex;`
- For synchronization:

```
pthread_mutex_lock(&myMutex)
//critical section code
pthread_mutex_unlock(&myMutex)
```
- After all work is done, need to destroy them
 - `pthread_mutex_destroy (&myMutex);`
- See example p2.c

CIT 595

27

Cond and Wait

- If we want one thread to signal an event to another we need to use Conditional variables
 - `pthread_cond_t condVariable`
- Idea is one thread wait until a certain condition is true
 - Test condition
 - If not true, calls `pthread_cond_wait(..)` to block until it is
- Another thread makes the condition true and call `pthread_cond_signal(...)` to unblock the thread waiting
- To avoid race conditions, the conditional variable must be use a mutex

CIT 595

28

Example

- Signalling thread

```
pthread_mutex_lock(&mutex);  
flag = 1;  
pthread_cond_signal(&condition);  
pthread_mutex_unlock(&mutex);
```

- condition is conditional variable
 - type pthread_cond_t
- mutex is mutex variable
 - type pthread_mutex_t

- Waiting thread

```
pthread_mutex_lock(&mutex);  
if(flag == 0)  
    pthread_cond_wait(&condition,  
                    &mutex);  
pthread_mutex_unlock(&mutex)
```

- Wait will automatically release the mutex while it waits
- After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread.
- Programmer is then responsible for unlocking *mutex* when the thread is finished with it

CIT 595

29

Deadlock in Resource Sharing Environment

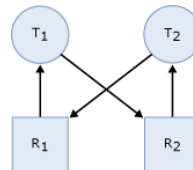
- A *deadlock* occurs when 2 or more processes/threads permanently block each other by each having a lock on a resource which the other tasks are trying to lock
- Deadlock can occur due to
 - Locks: Waiting to acquire locks on resources, such as objects, pages etc.
 - Sharing resources such as I/O devices printer, disks etc.

CIT 595

30

Deadlock High-Level View

- Task (thread/process) T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).



- Task T2 has a lock on resource R2 and has requested a lock on resource R1

- Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists

CIT 595

31

Deadlock Handling

- OS must also monitor or avoid deadlock
- Monitor (more practical approach):
 - Another service routine that periodically goes and checks on waiting tasks
 - If deadlock is determined, then
 - Abort all processes/threads within that are deadlock – expensive
 - Abort one process at a time till deadlock is cycle is eliminated
- Avoidance
 - By not explicitly over allocating resources
 - However this requires prior knowledge of total resources needed by each task – not practical
 - Difficult but many algorithms are proposed
 - E.g. Banker's Algorithm

CIT 595

32

Detection with only one instance of a resource type

- Construct a resource *graph*
- A graph is collection of nodes and edges
 - Node: process (circles) or resource (square)
 - Edge: Connects to nodes together with arrow indicating direction
 - Resource held by process (Resource -> Process)
 - Process wanting resource (Process -> Resource)
- If the graph contains one or more cycles then a deadlock exists
 - Use Depth First Search (DFS)
 - Start at the node and explore as far as possible along each branch before backtracking
 - If you see a node revisited again then we have found a cycle

CIT 595

33

Example

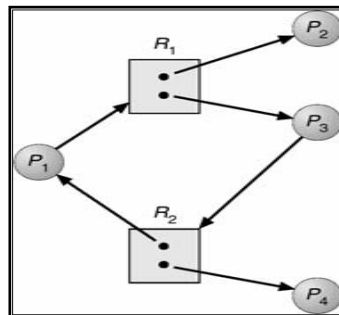
- Processes A – G & Resources R - W
 - A holds R and want S
 - B holds nothing but wants T
 - C holds nothing but wants S
 - D holds U and wants S and T
 - E holds T and wants V
 - F holds W and wants S
 - G holds V and wants U
- Search starting at B leads – B, T, E, V, G, U, D, T
 - See T twice, found a cycle

CIT 595

34

E.g. Detection with multiple instances of a resource type

- With more than one instances of resource type, a cycle can exist but deadlock may not



CIT 595

35

Deadlock Avoidance

- Grant a resource request only if request cannot lead to a deadlock either immediately or in the near future
 - No deadlock is known as *safe* state
- Conservative Approach:
 - Each process declares the maximum number of resource units of each resource class that it may require
 - OS processes each request, determines granting based on the current allocation and available resources leads to safe state
 - If safe state is possible then actually grant the request
- Safe State
 - Construct a sequence of process completion, resource release and resource allocation events through which each Process can obtain its max resources for each resource class

CIT 595

36

Bankers Algorithm

- Modeled after banker that gives credit to customers
- Idea is that not all customers will need their maximum credit immediately
 - Reserve lesser units than the total need
- Before actually giving credit check to see if financial safe

- Example with Single Resource class
 - System contains 10 units of resource class
 - Max Need for P1 = 8, P2 = 7, P3 = 5
 - Current Allocation for (P1 =3, P2 = 1, P3 = 3)
 - If process P1 makes request for one more resource unit i.e. (1, 0, 0) is the system in safe state?

CIT 595

37

Banker's Algorithm

- To complete a process P_i , check for all resource class R_k
 $Total_resource_k - Total_allocated_k \geq$
 $Max_need_{i,k} - Allocated_resources_{i,k}$
 - $Total_Allocated = \Sigma (Allocated_resources_{i,k})$

- If satisfied, it simulates completion of Process P_i and release of all resources allocated to it by updating $Total_allocated_k$ for each R_k

- P1's request for one resource unit leads system into safe state
 - Sequence P3, P1, P2

CIT 595

38